Operating Systems

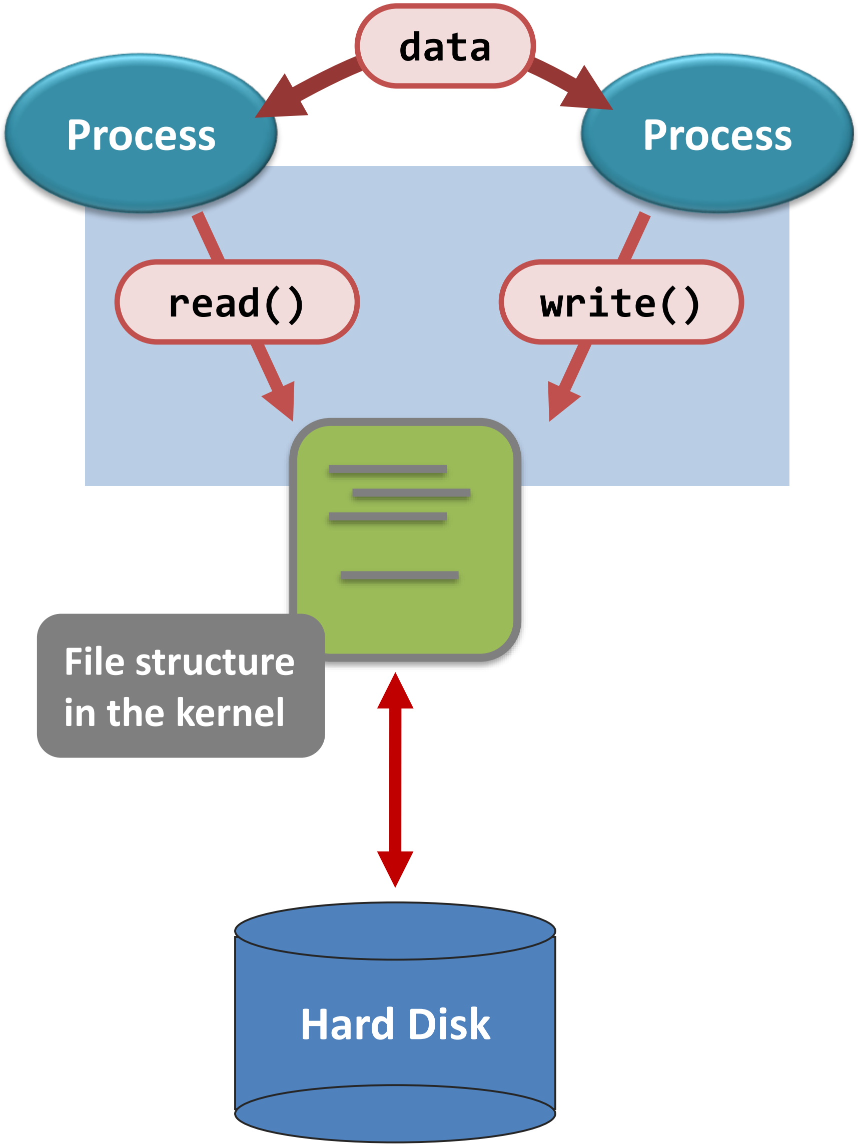
Prof. Yongkun Li
中国科大-计算机学院 教授
http://staff.ustc.edu.cn/~ykli

Ch5
Process Communication & Synchronization
-Part 2

IPC problem: Race condition

Evil source: the shared objects

- Pipe is implemented with the thought that **there may be more than one process accessing it "*at the same time*"**
- For shared memory and files, **concurrent access may yield <u>unpredictable outcomes</u>**



Understanding the problem…

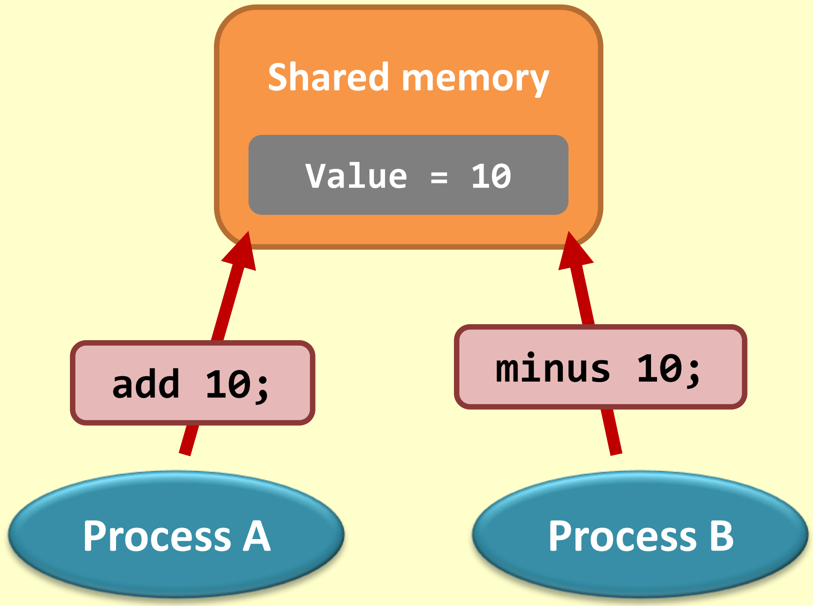
High-level language for Program A

```
attach to the shared memory X;
add 10 to X;
exit;
```

High-level language for Program B

```
attach to the shared memory X;
minus 10 to X;
exit;
```

The Scenario



Guess what the final result should be?

It may be 10, 0 or 20, can you believe it?

Understanding the problem…

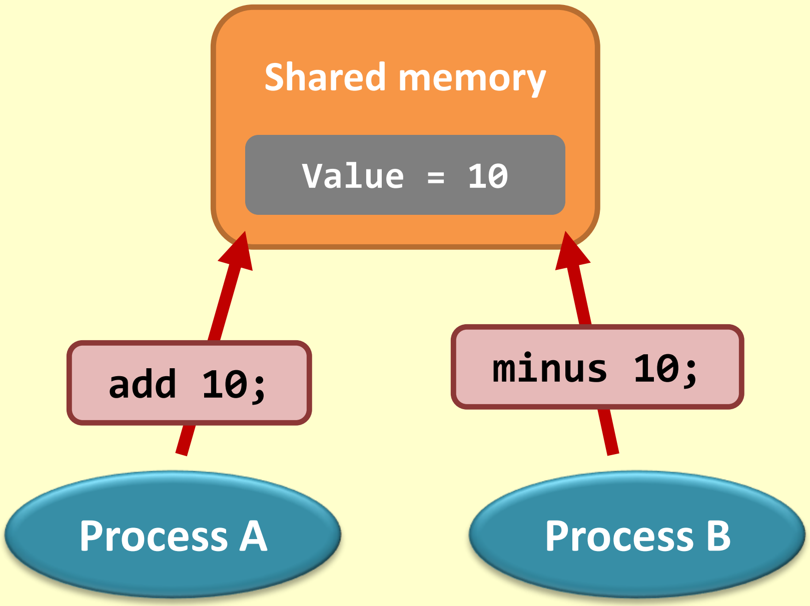
High-level language for Program A

```
attach to the shared memory X;
add 10 to X;
exit;
```

High-level language for Program B

```
attach to the shared memory X;
minus 10 to X;
exit;
```

The Scenario



Remember the flow of executing a program and the system hierarchy?

Understanding the problem…

High-level language for Program A

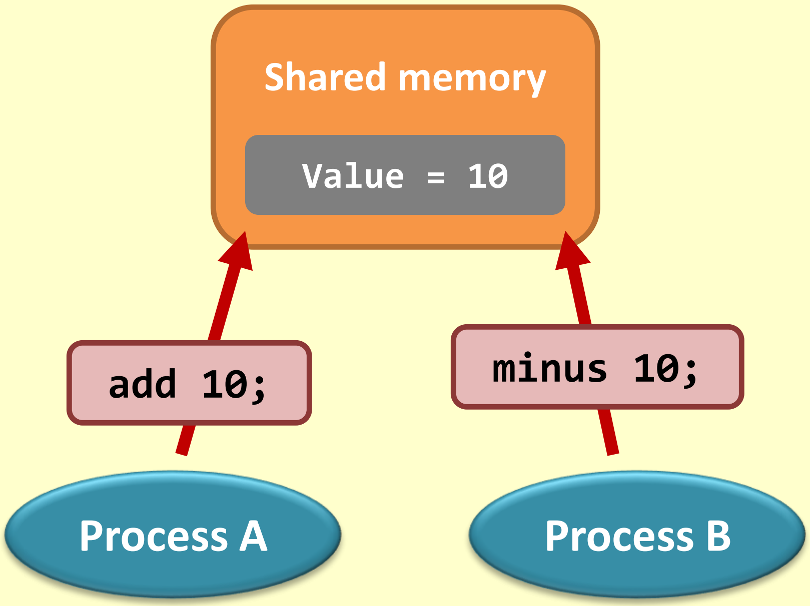
```
1   attach to the shared memory X;
2   add 10 to X;
3   exit;
```

This operation is not atomic

Partial low-level language for Program A

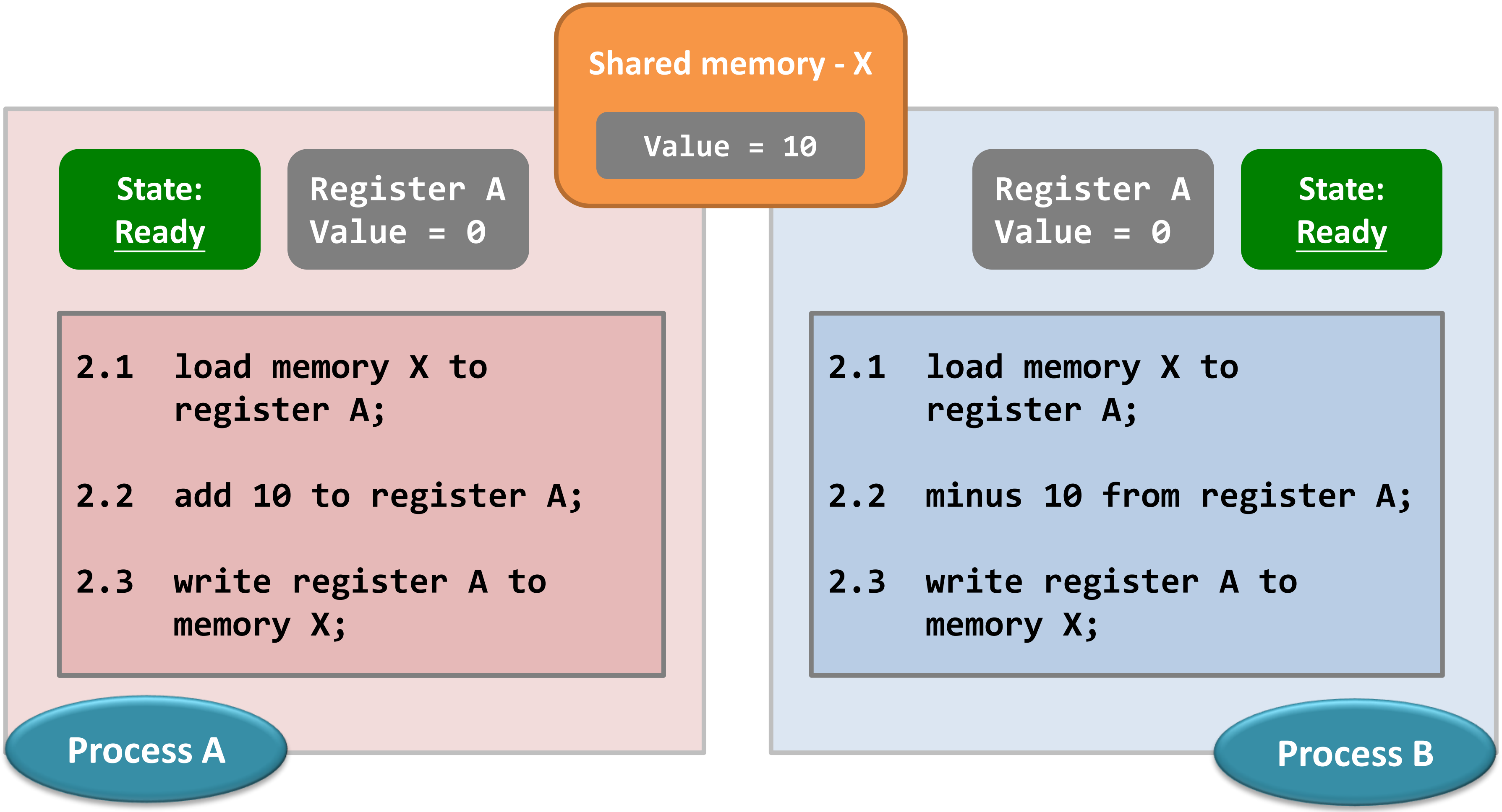
```
1     attach to the shared memory X;
......
2.1   load memory X to register A;
2.2   add 10 to register A;
2.3   write register A to memory X;
......
3     exit;
```

The Scenario



Guess what? This code block is evil!

Understanding the problem…



The initial setting

Execution Flow #1

Problem not yet arise...

Shared memory - X

Value = 10

State: Running

Register A Value = 10

Register A Value = 0

State: Ready

1

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

```
2.1   load memory X to
      register A;

2.2   minus 10 from register A;

2.3   write register A to
      memory X;
```

Process A

Process B

Execution Flow #1, Step 1

Problem not yet arise...

Shared memory - X

Value = 10

Process A

State: Running

Register A Value = 20

1 2.1 load memory X to register A;

2 2.2 add 10 to register A;

2.3 write register A to memory X;

Process B

Register A Value = 0

State: Ready

2.1 load memory X to register A;

2.2 minus 10 from register A;

2.3 write register A to memory X;

Execution Flow #1, Step 2

Problem not yet arise…

Shared memory - X

Value = 20

State: Running

Register A
Value = 20

```
1  2.1  load memory X to
        register A;
2  2.2  add 10 to register A;
3  2.3  write register A to
        memory X;
```

Process A

Register A
Value = 0

State: Ready

```
2.1  load memory X to
     register A;
2.2  minus 10 from register A;
2.3  write register A to
     memory X;
```

Process B

Execution Flow #1, Step 3

Problem not yet arise…

Shared memory - X

Value = 20

Process A

State: Ready

Register A Value = 20

1. 2.1 load memory X to register A;
2. 2.2 add 10 to register A;
3. 2.3 write register A to memory X;

Context Switching

Process B

State: Running

Register A Value = 20

4. 2.1 load memory X to register A;

2.2 minus 10 from register A;

2.3 write register A to memory X;

Execution Flow #1, Step 4

Problem not yet arise…

Shared memory - X

```
Value = 20
```

Process A

State: <u>Ready</u>

```
Register A
Value = 20
```

1 / 2.1 load memory X to register A;

2 / 2.2 add 10 to register A;

3 / 2.3 write register A to memory X;

Process B

```
Register A
Value = 10
```

State: <u>Running</u>

4 / 2.1 load memory X to register A;

5 / 2.2 minus 10 from register A;

2.3 write register A to memory X;

Execution Flow #1, Step 5

Problem not yet arise…

Shared memory - X

`Value = 10`

Process A

- **State:** <u>Ready</u>
- `Register A Value = 20`

```
1  2.1  load memory X to
        register A;
2  2.2  add 10 to register A;
3  2.3  write register A to
        memory X;
```

Process B

- `Register A Value = 10`
- **State:** <u>Running</u>

```
2.1  load memory X to       4
     register A;
2.2  minus 10 from register A;  5
2.3  write register A to    6
     memory X;
```

<u>Execution Flow #1, Step 6</u>

Execution Flow #2

Problem arise…

Shared memory - X

Value = 10

Process A

State: Running

Register A
Value = 10

1

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Process B

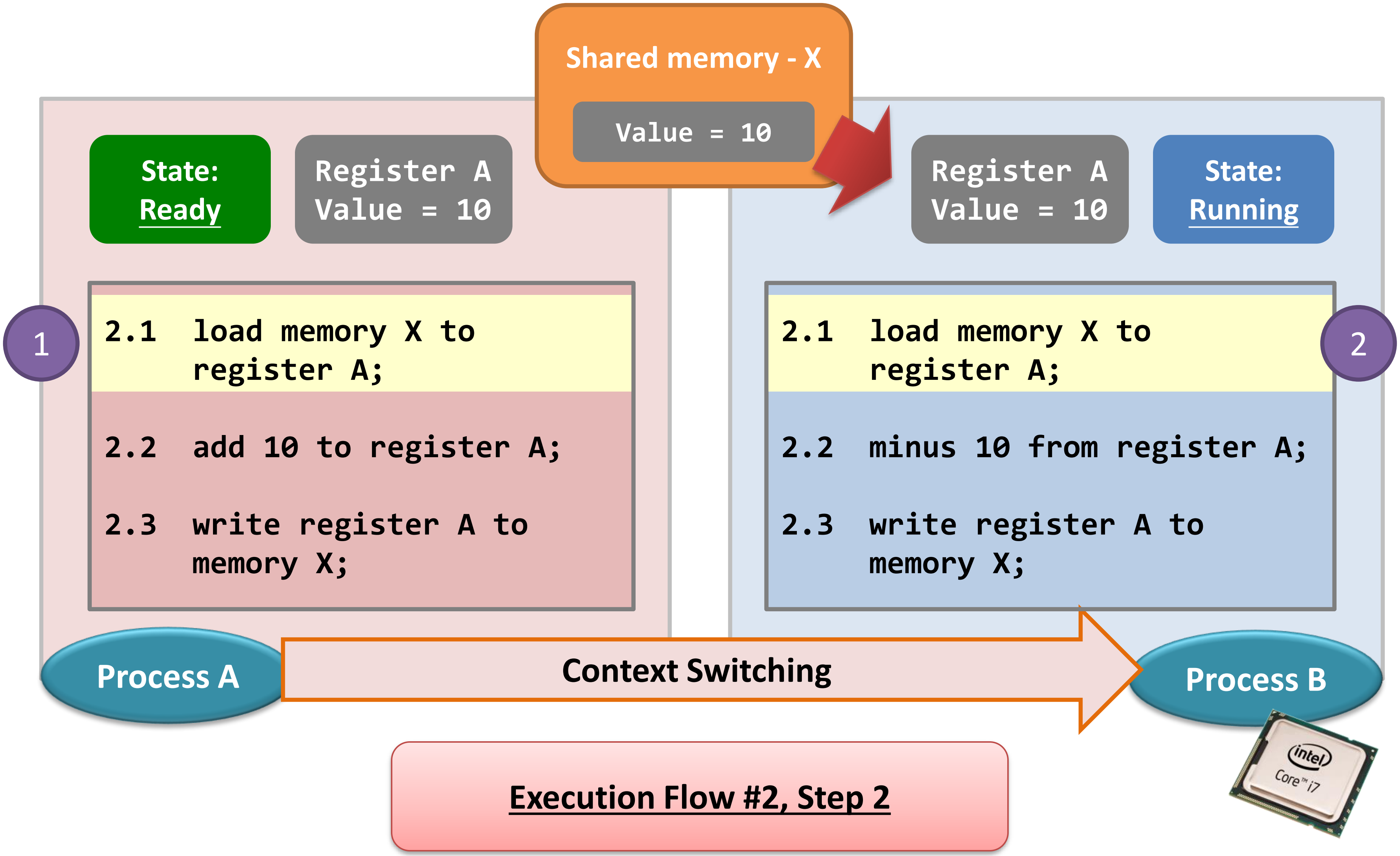
Register A
Value = 0

State: Ready

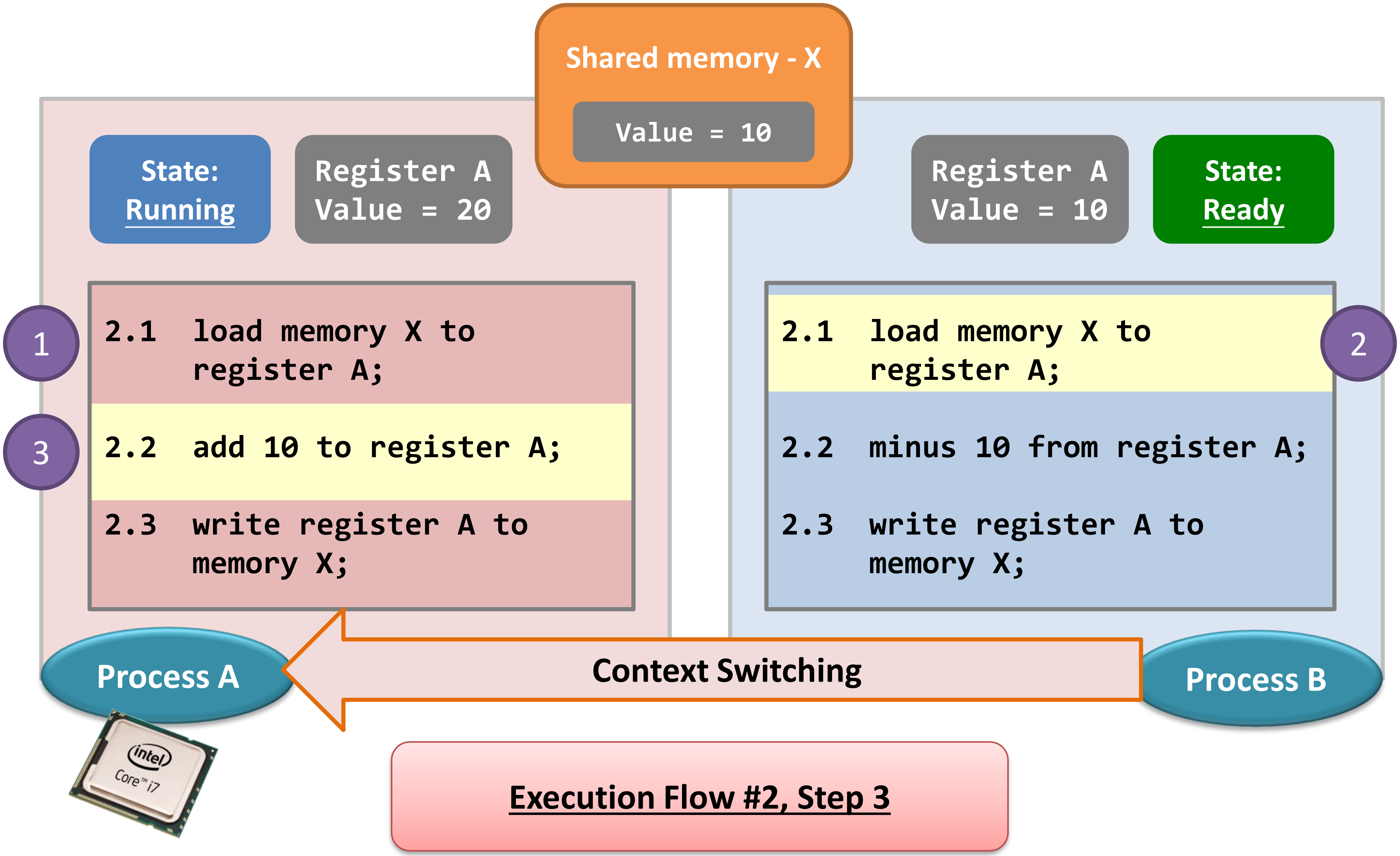
```
2.1   load memory X to
      register A;

2.2   minus 10 from register A;

2.3   write register A to
      memory X;
```

Execution Flow #2, Step 1

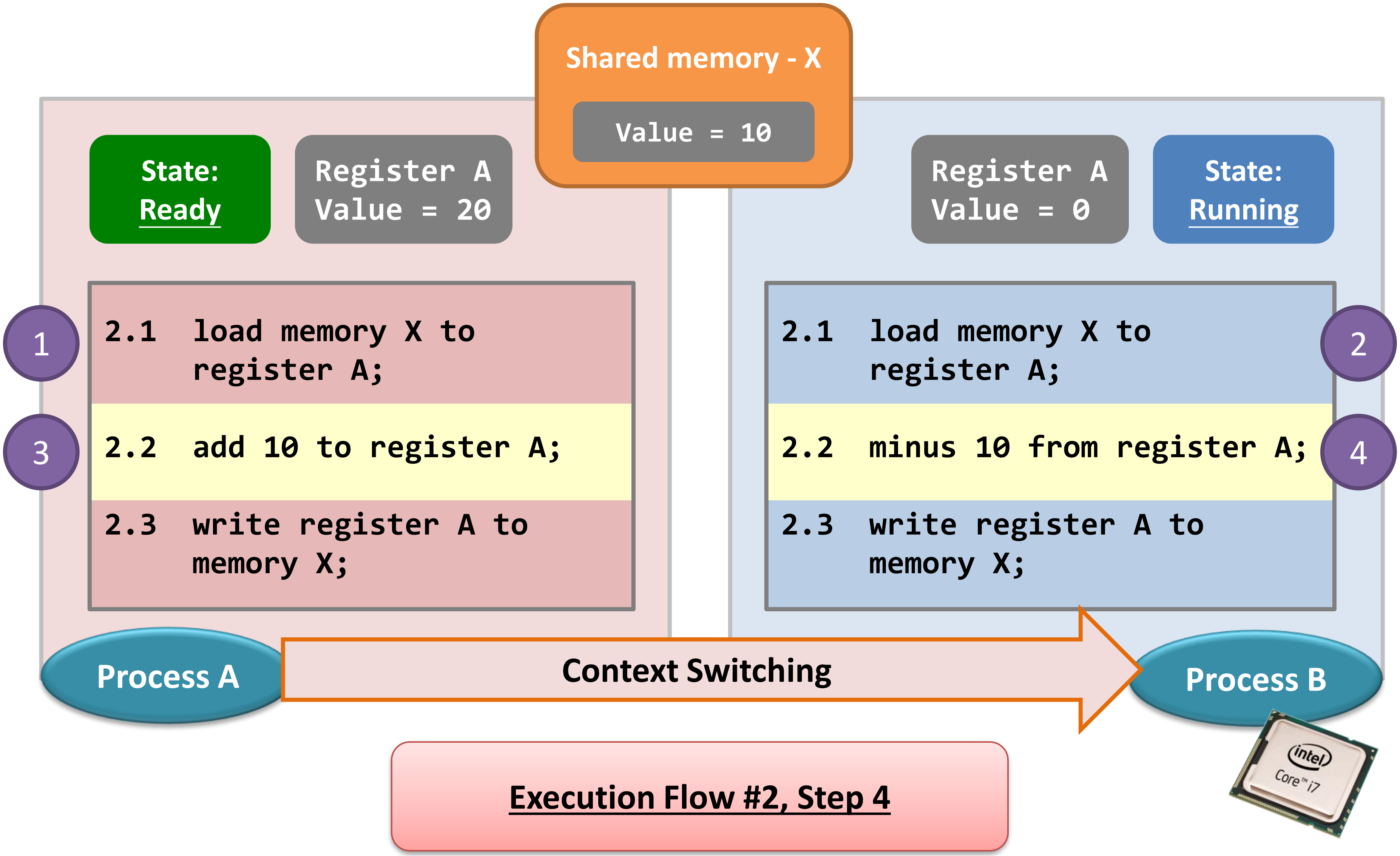
Problem arise…



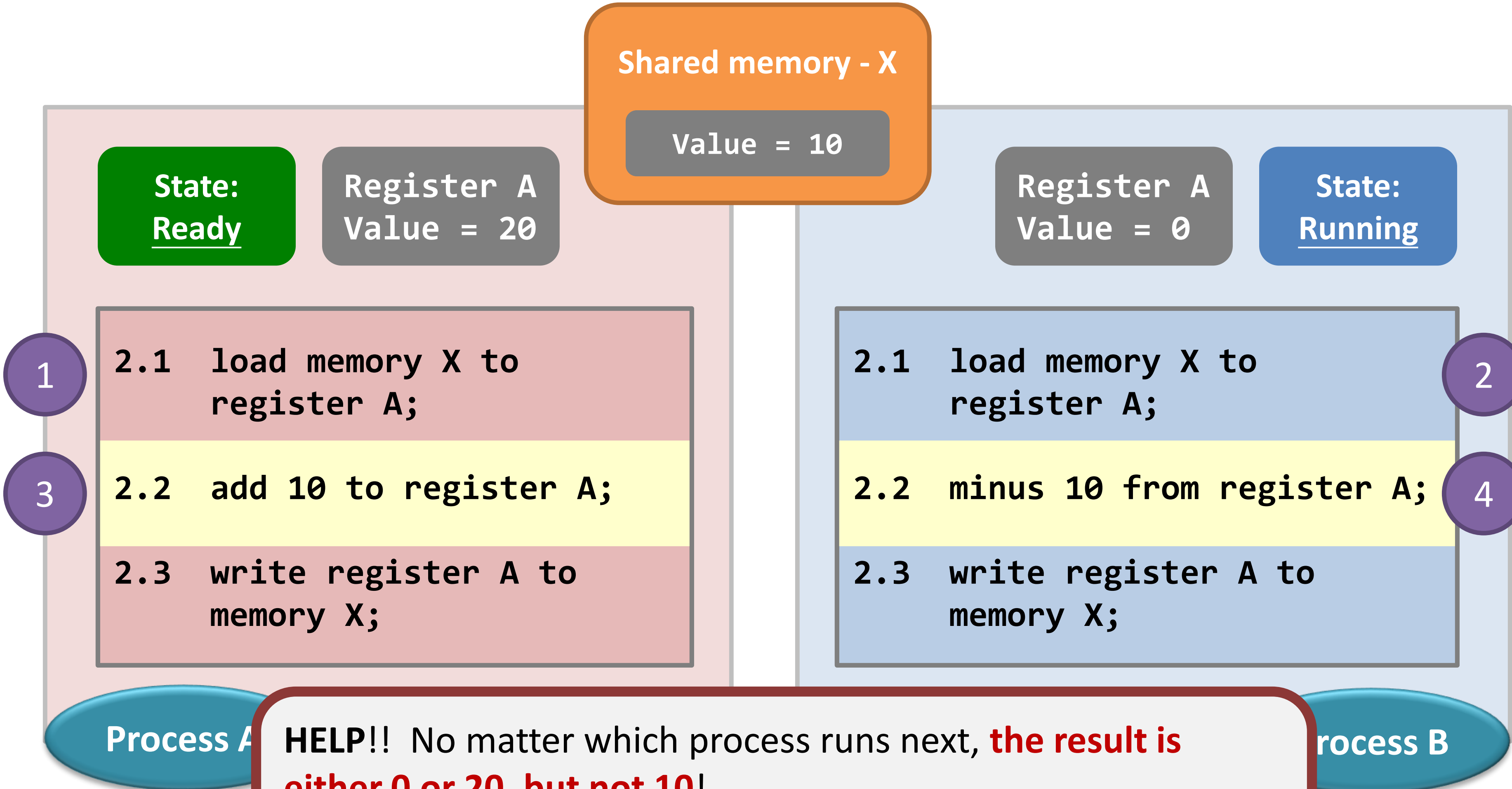
Problem arise…



Problem arise…



Problem arise...



HELP!! No matter which process runs next, **the result is either 0 or 20, but not 10**!

The final result depends on the execution sequence!

Race condition – the curse

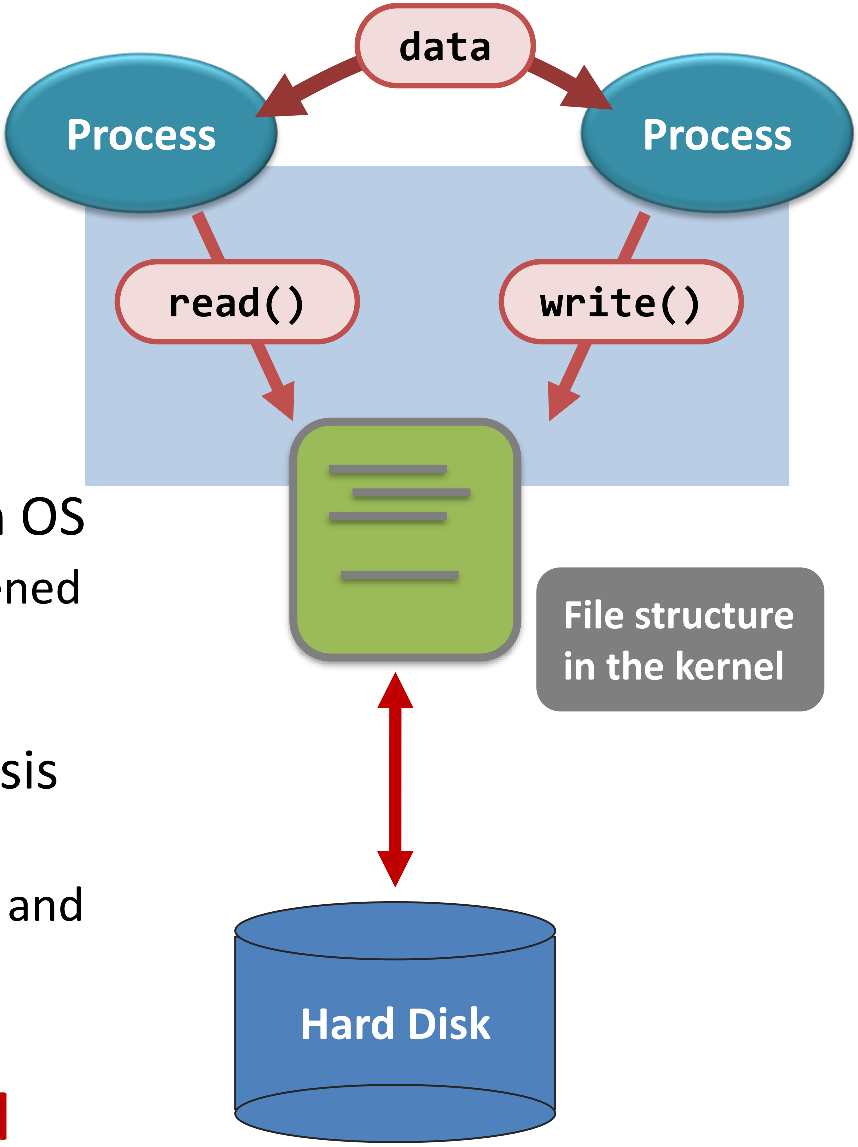
- The above scenario is called the **race condition**.

- A **race condition** means
  - the outcome of an execution depends on a particular order in which the shared resource is accessed.

- Remember: race condition is always a bad thing and debugging race condition has no fun at all!
  - It may end up …
    - 99% of the executions are fine.
    - 1% of the executions are problematic.

Race condition – the curse

- For shared memory and files, **concurrent access may yield unpredictable outcomes**
  - **Race condition**
- Common situation
  - Resource sharing occurs frequently in OS
    - EXP: Kernel DS maintaining a list of opened files, maintaining memory allocation, process lists…
  - Multicore brings an increased emphasis on multithreading
    - Multiple threads share global variables and dynamically allocated memory
- **Process synchronization is needed**



Topics in Process Synchronization

Cooperating Processes

concurrent accesses suffer from **race condition**

↓ Solution

Process Sychronization

Guarantee **mutual exclusion**

↓ Application

Semaphore Usage

Avoid **deadlock**

Idea: How to achieve

Define **critical section**

How to implement

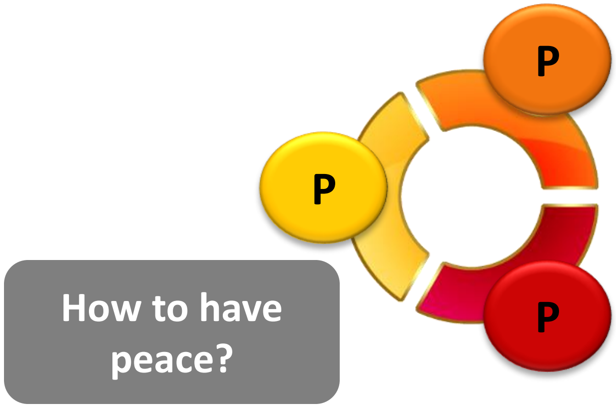
- Four requirements
- Software-based proposals
  - Disabling interrupts
  - strict alternation
  - peterson's solution
  - mutex lock
  - **Semaphore (best choice)**

Classic problems

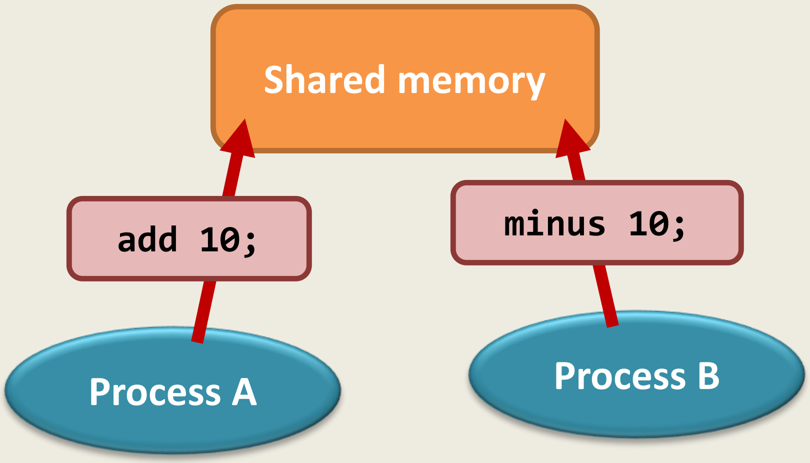
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Inter-process communication (IPC)

- Mutual exclusion
  - what & how to achieve?



Mutual Exclusion



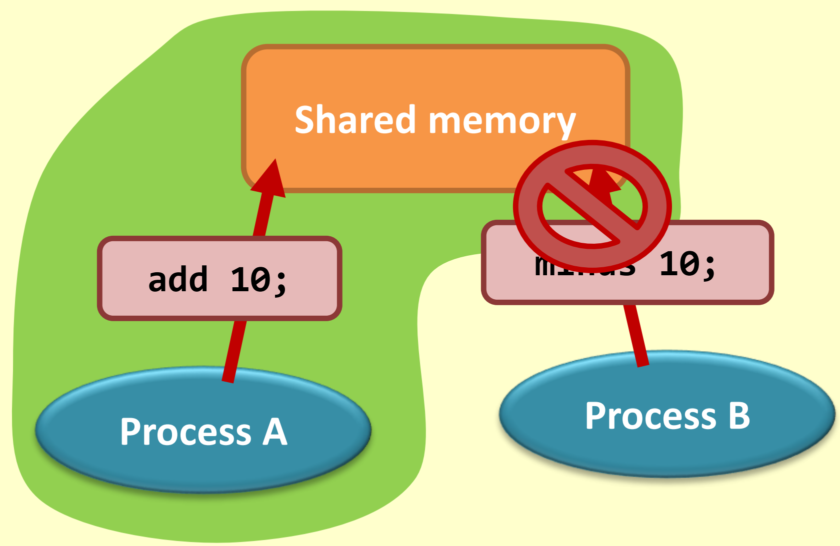
Two processes playing with **<u>the same shared memory</u>** is dangerous.

We will face the curse - **race condition**.

The solution can be simple:

When I'm playing with the shared memory, no one could touch it.

This is called **mutual exclusion**.
A set of processes would not have the problem of race condition *if mutual exclusion is guaranteed*.

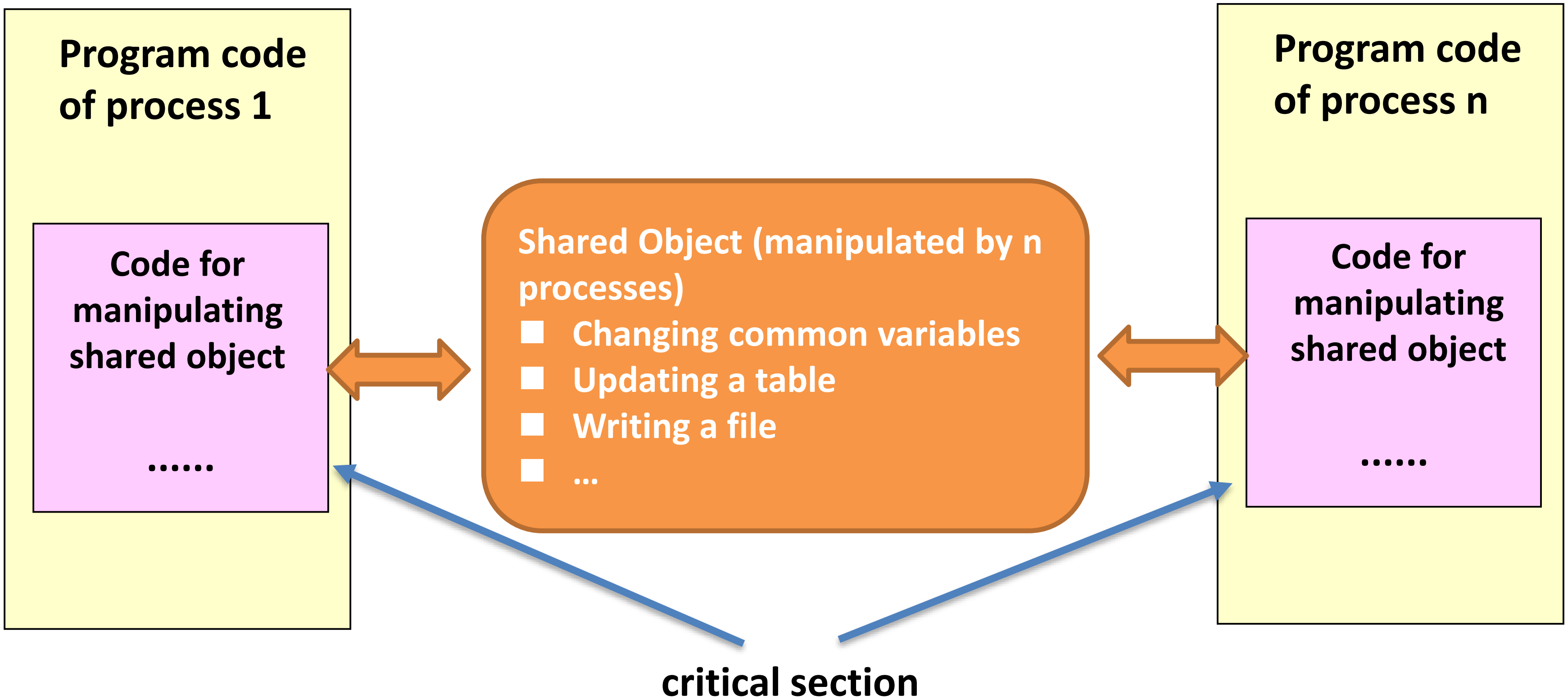


How to realize mutual exclusion?

- Kernel
  - Preemptive kernels and nonpreemptive kernels
    - Allows (not allow) a process to be preempted while it is running in kernel mode
  - A nonpreemptive kernel is essentially free from race conditions on kernel data structures, and also easy to design (especially for SMP architecture)
  - Why would anyone favor a preemptive kernel
    - More responsive
    - More suitable for real-time programming

Mutual Exclusion

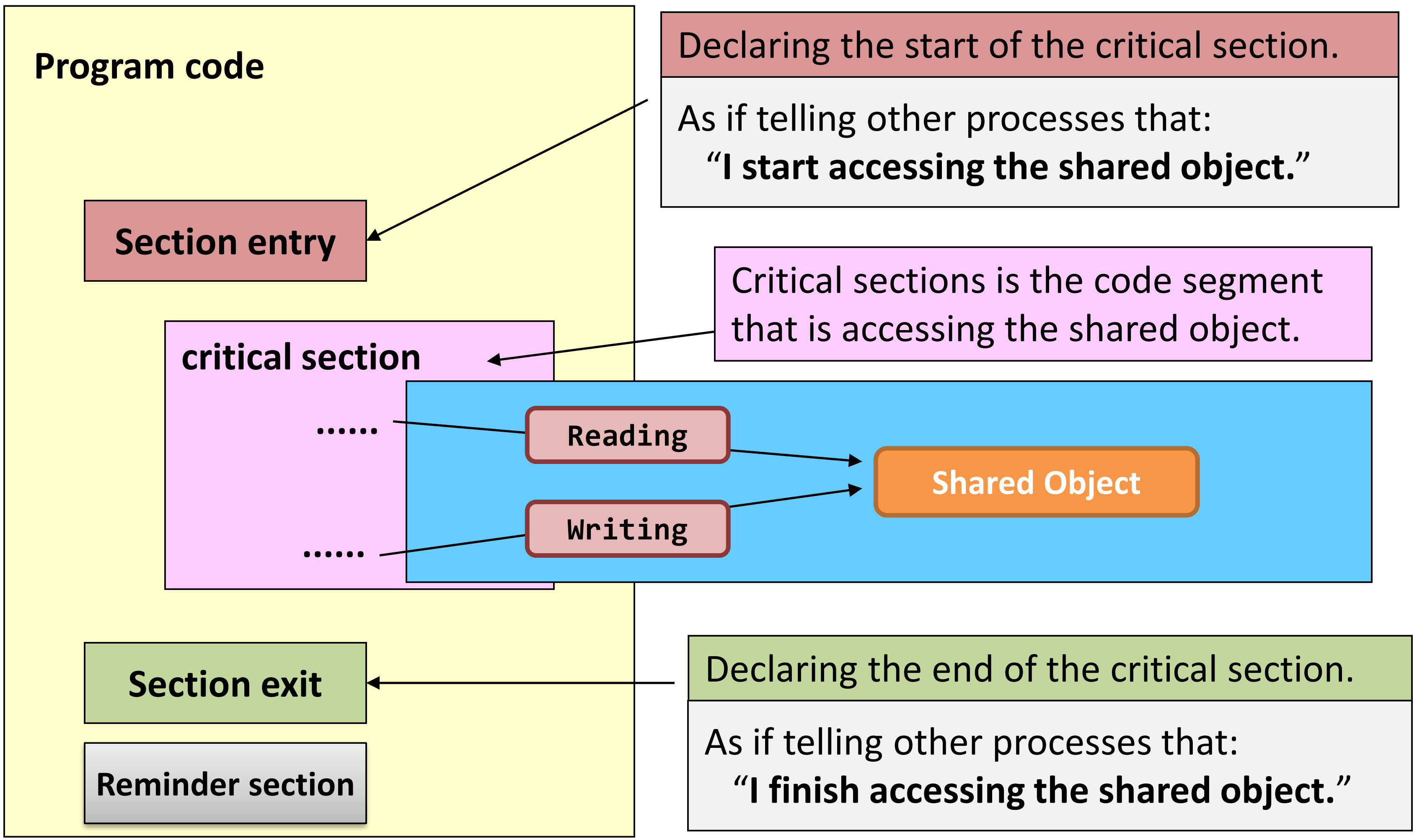
- More generally, how to realize?



Solution: To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.

Critical Section – General Structure

To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.



Critical Section – Example

Process A

Need a section entry here

Critical Section

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Need a section exit here

Process B

Need a section entry here

```
2.1   load memory X to
      register A;

2.2   minus 10 from register A;

2.3   write register A to
      memory X;
```

Need a section exit here

Important concept here.

Both regions are called critical sections, yet they can be different.

Summary…for the content so far…

- **Race condition** is a problem.
  - It makes a concurrent program producing **unpredictable** results if you are using shared objects as the communication medium.
  - The outcome of the computation **totally depends on the execution sequences** of the processes involved.

- **Mutual exclusion** is a requirement.
  - If it could be achieved, then the problem of the race condition would be gone.
  - Mutual exclusion hinders the performance of parallel computations.

Summary…for the content so far…

- **Defining critical sections** is a solution.
  - They are code segments that access shared objects.

  - Critical section must be **as tight as possible**.
    - Well, you can <u>declare the entire code of a program to be a big critical section</u>.
    - But, the program will be a very high chance to <u>block other processes</u> or to <u>be blocked by other processes</u>.

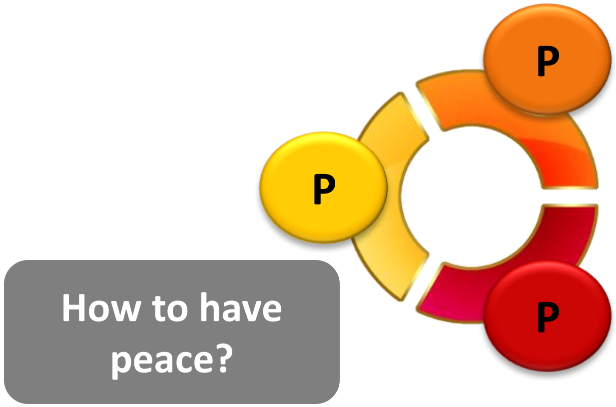
  - Note that **<u>one critical section</u>** can be designed for **accessing more than one shared objects**.

Summary…for the content so far…

- **Implementing section entry and exit** is a challenge.
  - The entry and the exit are **the core parts that guarantee mutual exclusion**, but not the critical section.
  - Unless they are correctly implemented, race condition would appear.

Inter-process communication (IPC)

- Mutual exclusion:
  - how to achieve?
  - how to implement? (section entry and exit)



Entry and exit implementation - requirements

- **Requirement #1: Mutual Exclusion**. No two processes could be simultaneously inside their critical sections.

> **Implication**: when one process is inside its critical section, any attempts to go inside the critical sections by other processes are not allowed.

- **Requirement #2**. Each process is executing at a nonzero speed, but no assumptions should be made about the relative speed of the processes and the number of CPUs.

> **Implication**: the solution **cannot depend on the time spent inside the critical section**, and the solution cannot assume the number of CPUs in the system.

Entry and exit implementation - requirements

- **Requirement #3: progress**. No process running outside its critical section should block other processes.

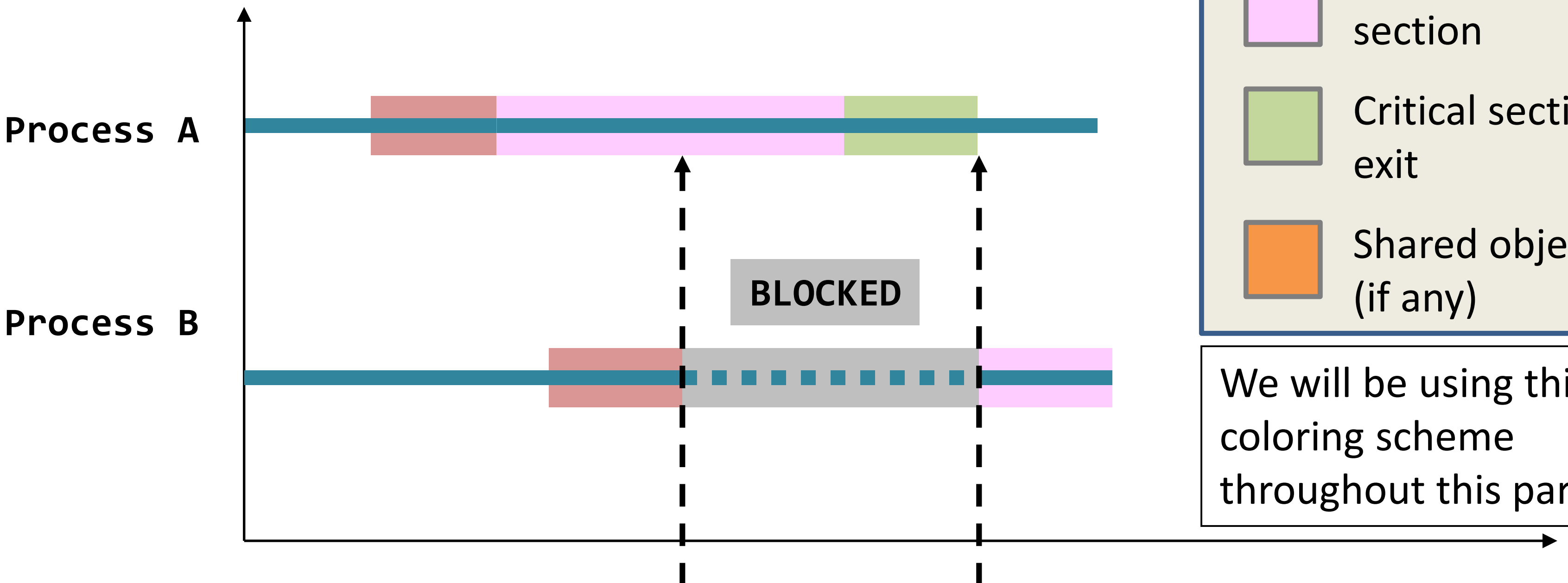
> **Implication**: Only processes that are not executing in their reminder sections can participate in deciding which will enter its critical section.

- **Requirement #4: Bounded waiting**. No process would have to wait forever in order to enter its critical section.

> **Implication**: There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section (no processes should be **starved to death**).

A typical mutual exclusion scenario

Remember, it is always the entry blocks other processes, but not the critical section.



Keys

- Critical section entry
- Inside Critical section
- Critical section exit
- Shared object (if any)

We will be using this coloring scheme throughout this part.

```
B tries to enter its critical
section but A is in its
critical section.
```

```
A leaves its critical section
and B resumes execution
accordingly.
```

Mutual Exclusion Implementation

- Challenges of Implementing **section entry** & **exit**
  - Both operations must be atomic
  - Also need to satisfy the above requirements
  - Performance consideration

- Hardware solution
  - Rely on atomic instructions
  - test_and_set()
  - compare_and_swap

Example: test_and_set()

- Definition

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

- Mutual exclusion implementation

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Example: compare_and_swap()

- Definition

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

- Mutual exclusion implementation

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

How to satisfy bounded waiting?

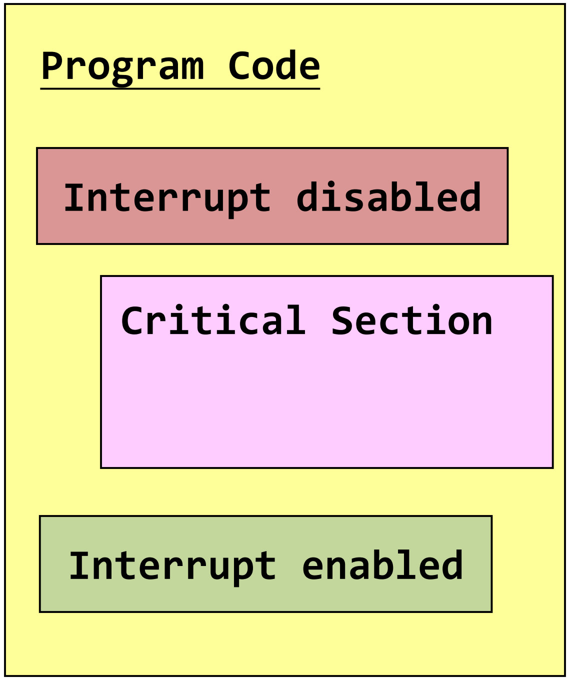
Enhanced version

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

lock is initialized as false

Proposal #1 – disabling interrupt.

- **Method**
  - Similar idea as nonpreemptive kernels
  - To **disable context switching** when the process is inside the critical section.
- **Effect**
  - When a process is in its critical section, no other processes could be able to run.
- **Implementation**
  - A new system call should be provided.
- **Correctness?**
  - **Correct,** but it is not an *attractive* solution.
  - Not as feasible in a multiprocessor environment
  - Performance issue (may sacrifice concurrency)



Proposal #2: Mutex Locks

- **Idea**
  - A process must acquire the lock before entering a critical section, and release the lock when it exits the critical section
  - Using a new shared object to detect the status of other processes, and "**lock**" the shared object

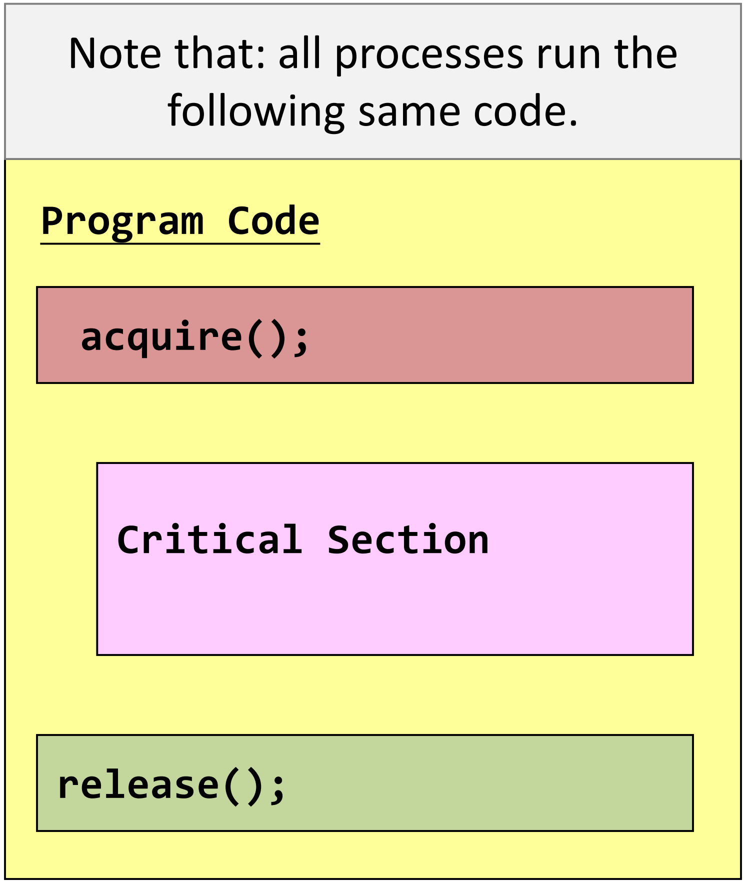
Shared object: "available" (lock)

```
acquire(){
   while(!available)
         ; /* busy waiting */
   available = false;
 }
```

```
release(){
   available = true;
 }
```

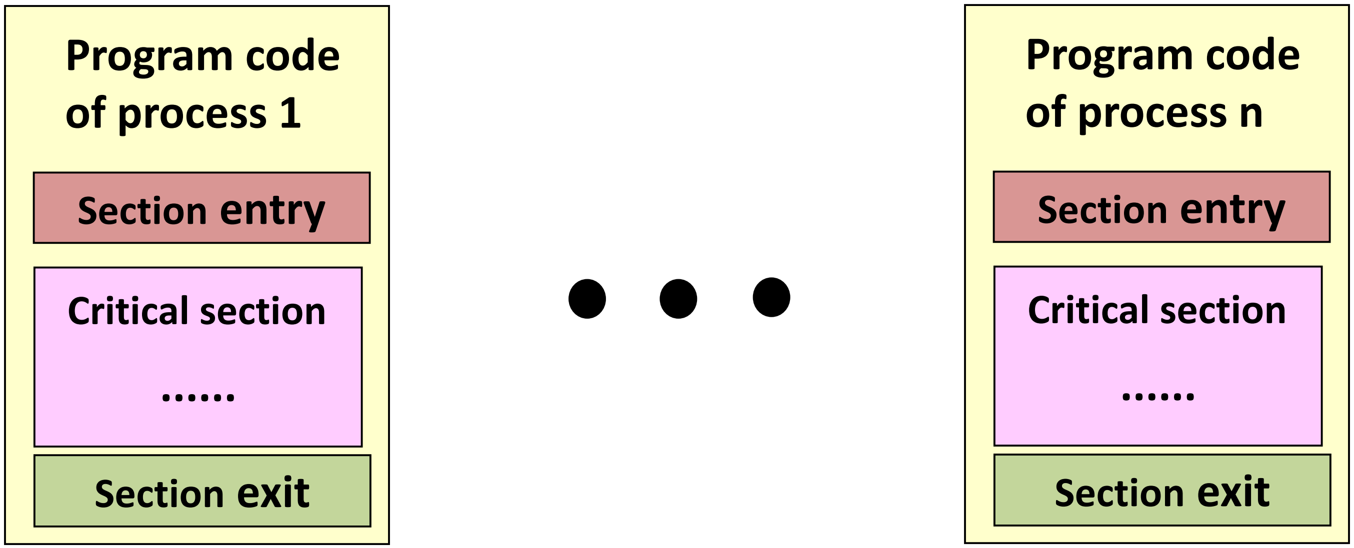
Proposal #2: Mutex Locks

- **Implementation**
  - Calls to acquire and release locks must be performed **atomically**
  - Often use hardware instructions
- **Issue**
  - Busy waiting: Waste CPU resource
    - **Spinlock**
- **Applications**
  - Multiprocessor system
    - When locks are expected to be held for short times



Other software-based solutions

- Aim
  - To decide which process could go into its critical section



- Key Issue
  - Detect the status of processes (section entry)
    - Need other shared variables

Proposal #3: Strict alternation

- **Method**
  - Using a new shared object to detect the status of other processes

Shared object "turn" — `initial Value = 0`

Process 0 (Allow to enter when turn == 0)

```
while (TRUE) {
    while( turn != 0 )
       ;  /* busy waiting */
    critical_section();
    turn = 1;
    non_critical_section();
}
```

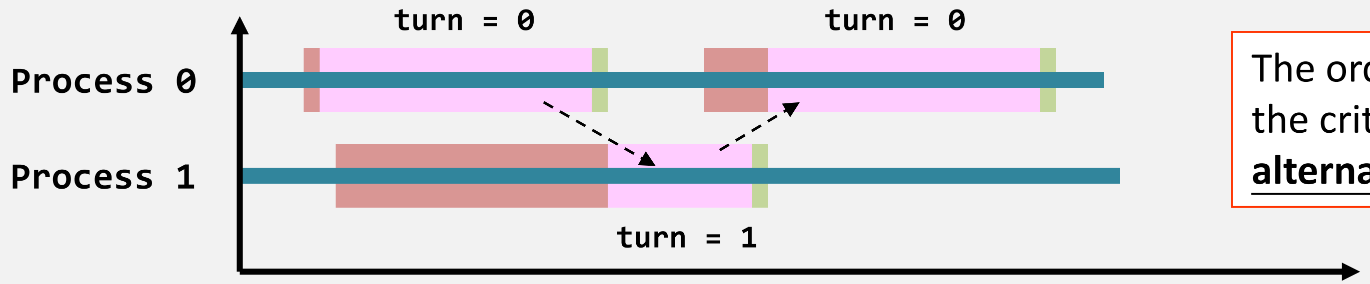
Entry

Exit

Process1 (Allow to enter when turn == 1)

```
while (TRUE) {
    while( turn != 1 )
       ;  /* busy waiting */
    critical_section();
    turn = 0;
    non_critical_section();
}
```

Proposal #3: Strict alternation



The order of executing the critical section is **alternating**.

Shared object "turn" `initial Value = 0`

Process 0

```
while (TRUE) {
    while( turn != 0 )
      ; /* busy waiting */
    critical_section();
    turn = 1;
    non_critical_section();
}
```

Process1

```
while (TRUE) {
    while( turn != 1 )
      ; /* busy waiting */
    critical_section();
    turn = 0;
    non_critical_section();
}
```

Proposal #3: Strict alternation - Cons

- Strict alternation seems good, yet, it is **inefficient**.
  - Busy waiting wastes CPU resources.

- In addition, the alternating order is **too strict**.
  - What if Process 0 wants to enter the critical section **twice in a row**? **NO WAY!**
  - Violate any requirement?

Requirement #3. No process running outside its critical section should block other processes.

Proposal #4: Peterson's solution

- How to improve the strict alternation proposal?

- The Peterson's solution:
  - Processes would act as a gentleman: if you want to enter, I'll let you first
  - No alternation is there
  - Share two data items
    - **`int turn;`** //whose turn to enter its critical section
    - **`Boolean interested[2];`** //if a process wants to enter

Proposal #4: Peterson's solution

Shared object: "turn" & "interested[2]"

```
int turn;                           /* who can enter critical section */
int interested[2] = {FALSE,FALSE};  /* wants to enter critical section*/

void enter_region( int process ) {  /* process is 0 or 1 */
  int other;                        /* number of the other process */
  other = 1-process;                /* other is 1 or 0 */
  interested[process] = TRUE;       /* want to enter critical section */
  turn = other;
  while ( turn == other &&
           interested[other] == TRUE )
      ;    /* busy waiting */
 }

void leave_region( int process ) {    /* process: who is leaving */
  interested[process] = FALSE;    /* I just left critical region */
 }
```

Entry

Exit

Proposal #4: Peterson's solution

```
1   int turn;
2   int interested[2] = {FALSE,FALSE};
3
4   void enter_region( int process ) {
5     int other;
6     other = 1-process;
7     interested[process] = TRUE;
8     turn = other;
9     while ( turn == other &&
              interested[other] == TRUE )
10        ;    /* busy waiting */
11  }
12
13  void leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

Line 8 akes the other one the turn to run.

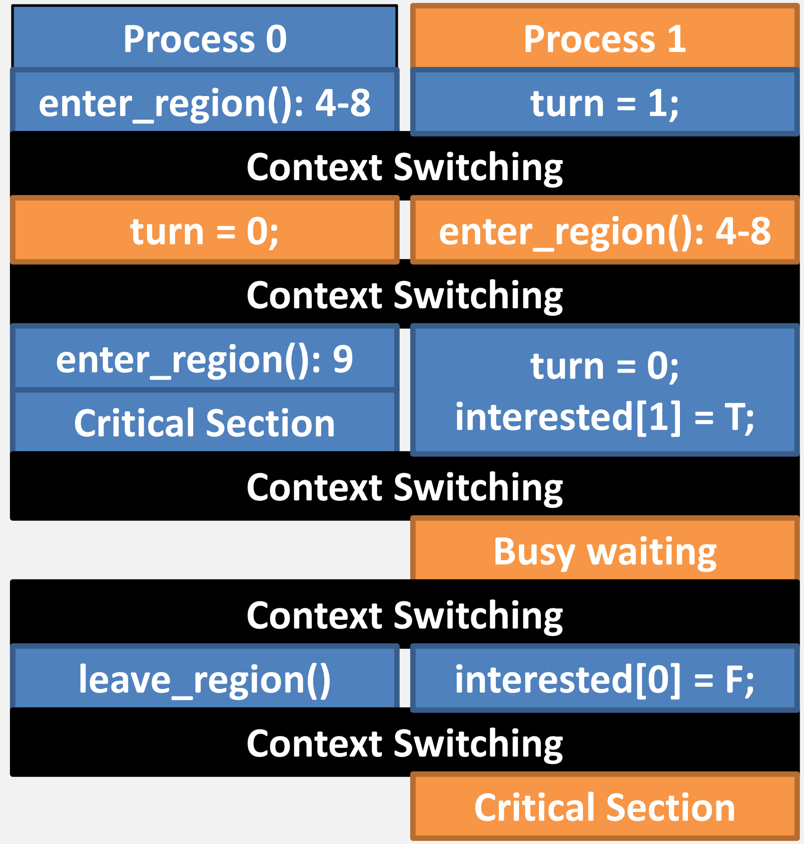
Of course, the process is willing to wait when she wants to enter the critical section.

<u>"I'm a gentleman!"</u>

The process always let another process to enter the critical region first although she wants to enter too.

Proposal #4: Peterson’s solution

```
1  int turn;
2  int interested[2] = {FALSE,FALSE};
3
4  void enter_region( int process ) {
5    int other;
6    other = 1-process;
7    interested[process] = TRUE;
8    turn = other;
9    while ( turn == other &&
             interested[other] == TRUE )
10       ;    /* busy waiting */
11 }
12
13 void leave_region( int process ) {
14   interested[process] = FALSE;
15 }
```

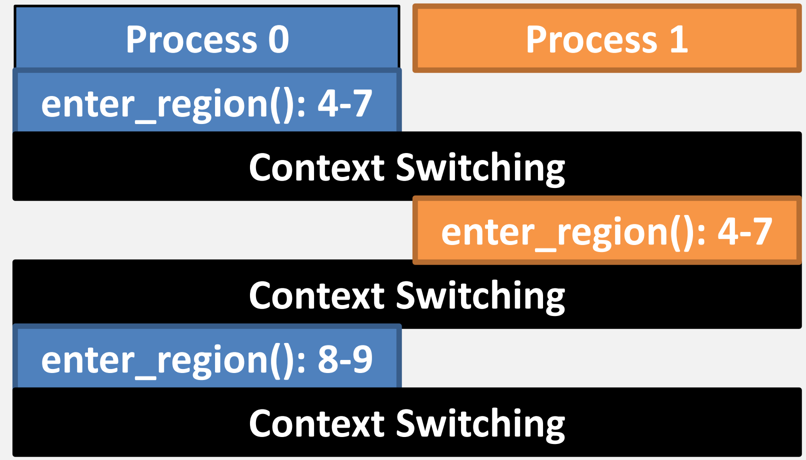


and the story goes on…

Can you show that the requirements are satisfied?

Proposal #4: Peterson’s solution

```
 1  int turn;
 2  int interested[2] = {FALSE,FALSE};
 3
 4  void enter_region( int process ) {
 5    int other;
 6    other = 1-process;
 7    interested[process] = TRUE;
 8    turn = other;
 9    while ( turn == other &&
                interested[other] == TRUE )
10      ;    /* busy waiting */
11  }
12
13  void leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

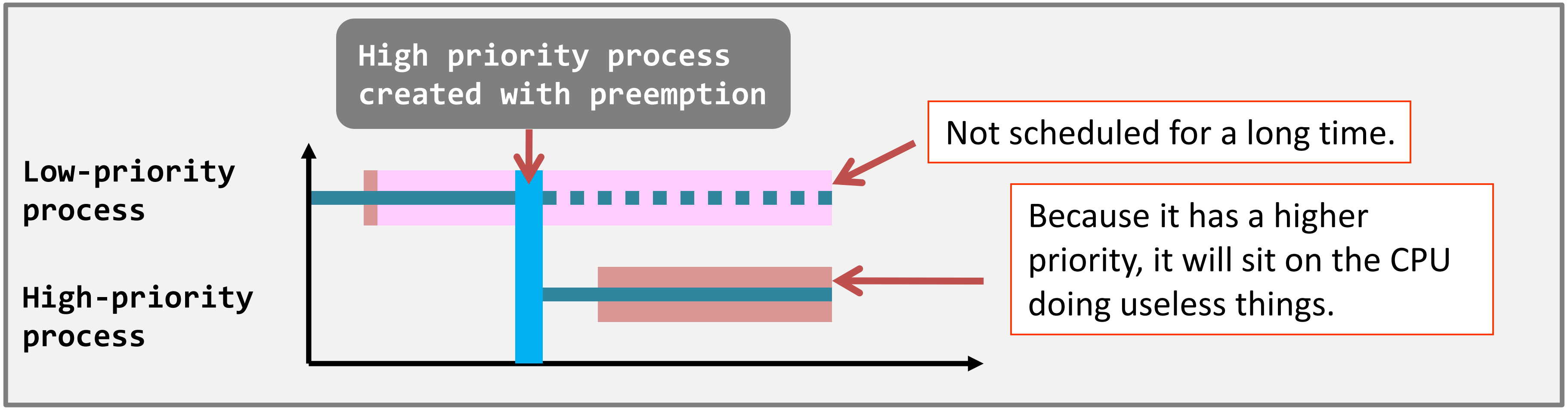


Can you complete the flow?
(what is the difference?)

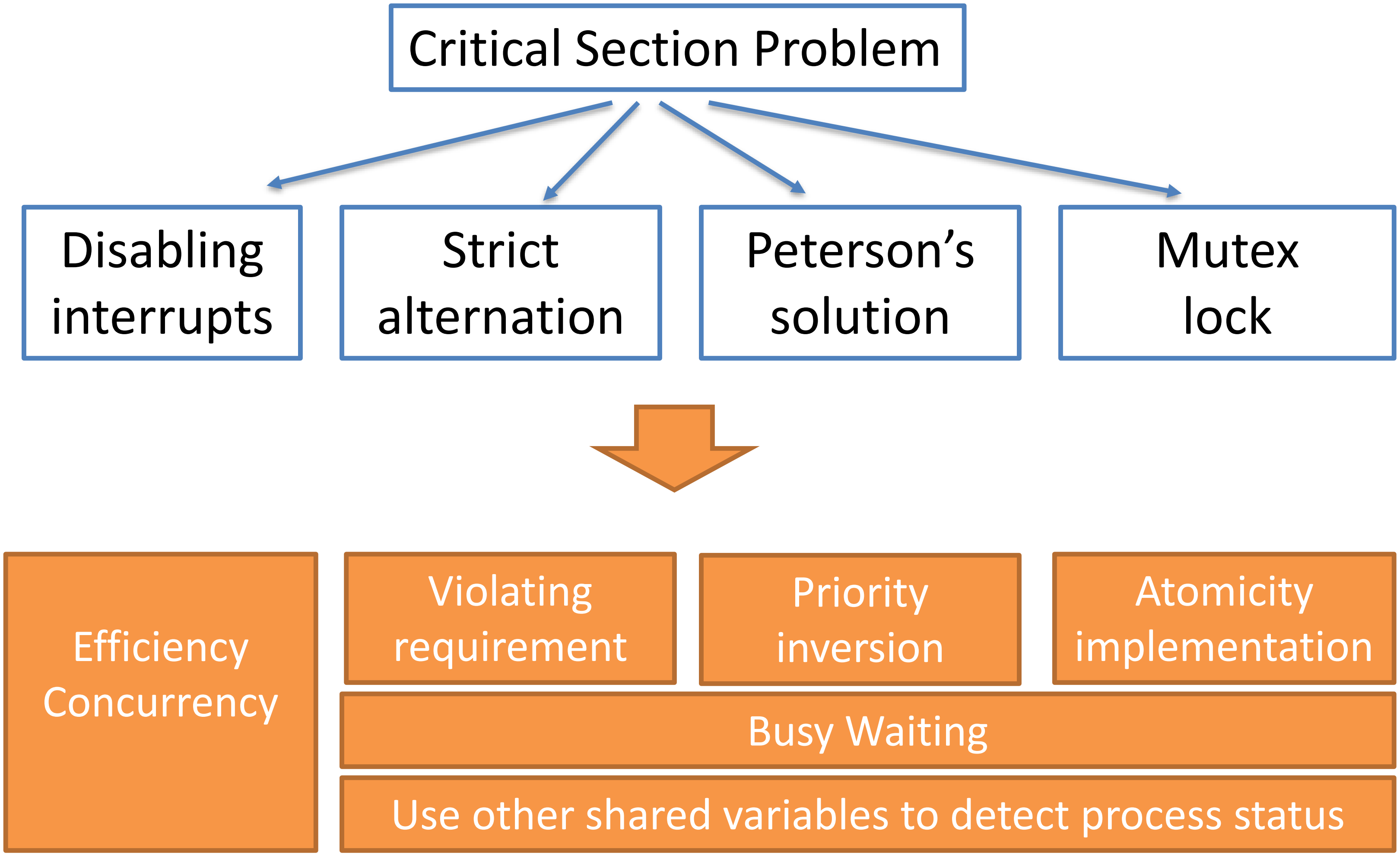
Can both processes progress?

Proposal #4: Peterson's solution – issues

- Busy waiting has its own problem…
  - **An apparent problem**: wasting CPU time.
  - **A hidden, serious problem**: **priority inversion problem.**
    - A low priority process is inside the critical region, but …
    - A high priority process wants to enter the critical region.
    - Then, the high priority process will perform busy waiting for a long time or even forever.

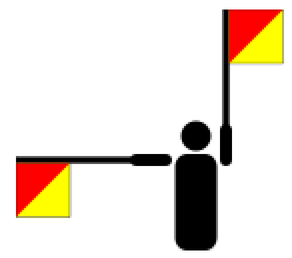


Story so far…


Critical Section Problem
Disabling interrupts
Strict alternation
Peterson’s solution
Mutex lock
Efficiency Concurrency
Violating requirement
Priority inversion
Atomicity implementation
Busy Waiting
Use other shared variables to detect process status

Final proposal: Semaphore

- In real life, semaphore is a flag signaling system.
  - It tells a train driver (or a plane pilot) when to stop and when to proceed.



source: wikipedia.

- When it comes to programming…
  - A semaphore is a data type.
  - You can imagine that it is **<u>an integer</u>** (but it is certainly not an integer when it comes to real implementation).

Final proposal: Semaphore

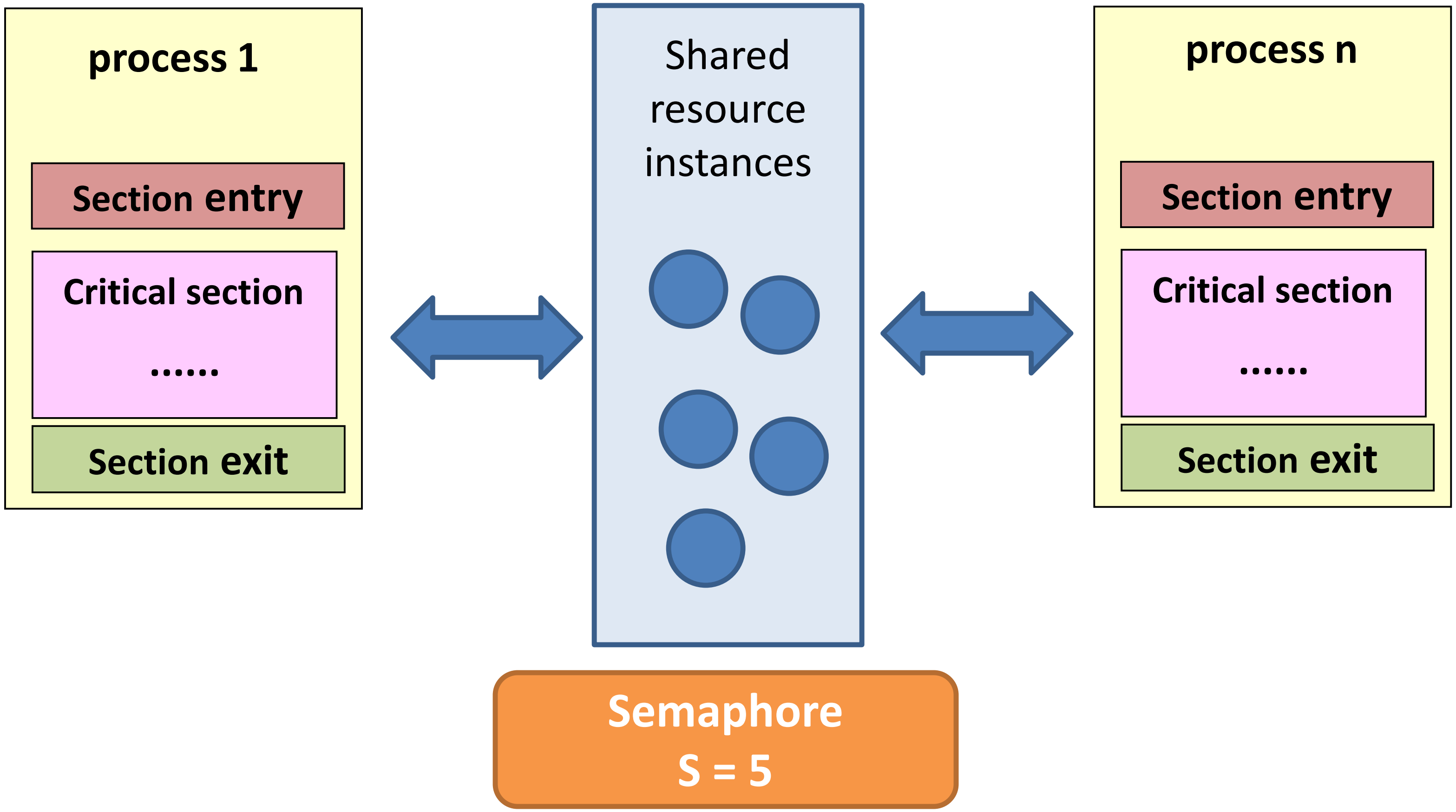
- Semaphore is a data type (**additional shared object**)
  - Denote the status or the number of resources
  - Two types
    - **Binary semaphore**: 0 or 1 (similar to mutex lock)
    - **Counting semaphore**: control finite number of resources

- Accessed through two standard **atomic** operations
  - **`down()`:** originally termed P (from Dutch *proberen*, "to test"), **`wait()`** in textbook
    - Decrementing the count
  - **`up()`:** originally termed V (from *verhogen*, "to increment"), **`signal()`** in textbook
    - Incrementing the count

Final proposal: Semaphore

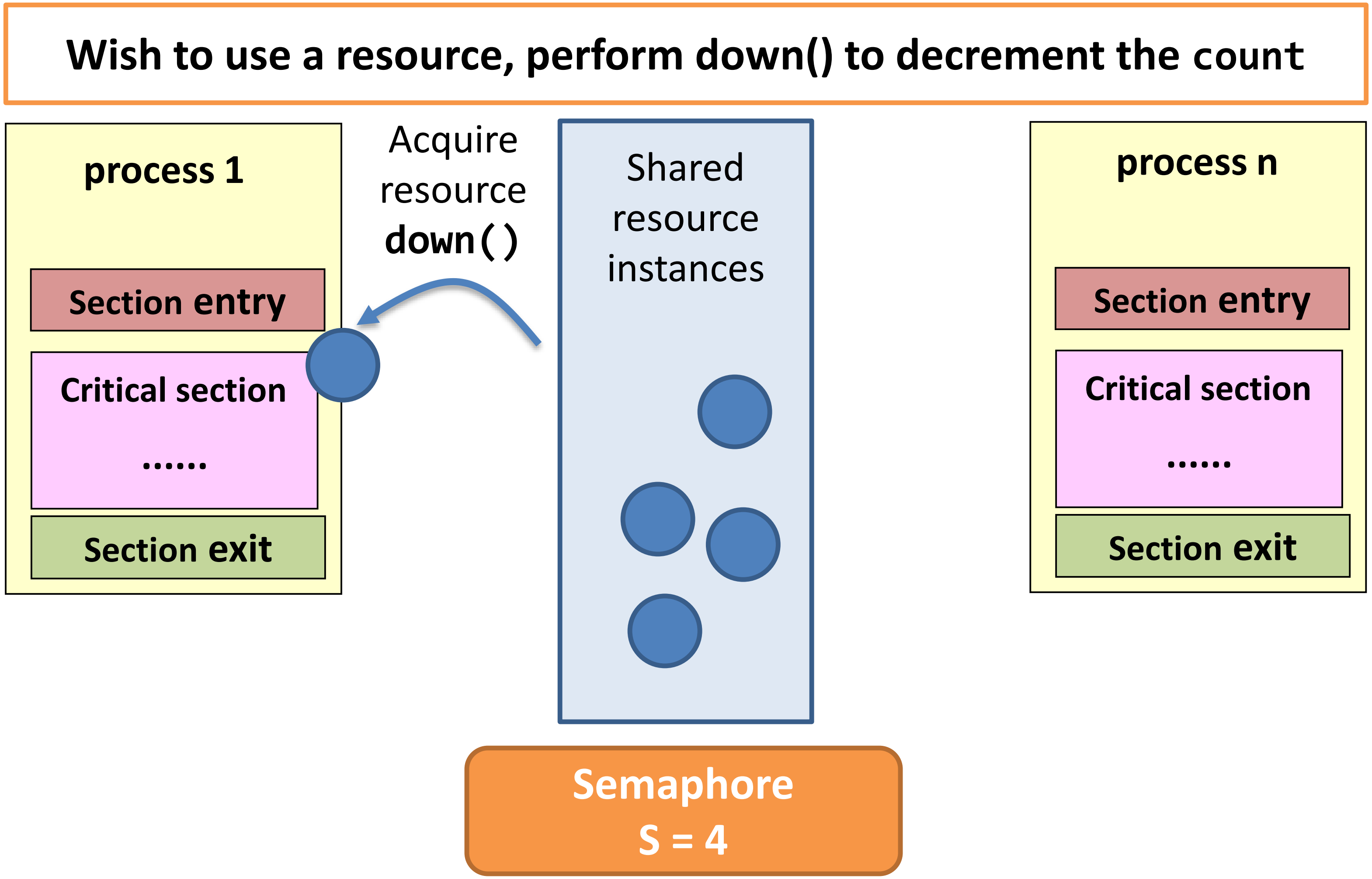
- Idea

Initialize the semaphore to the number of resource instances



Final proposal: Semaphore

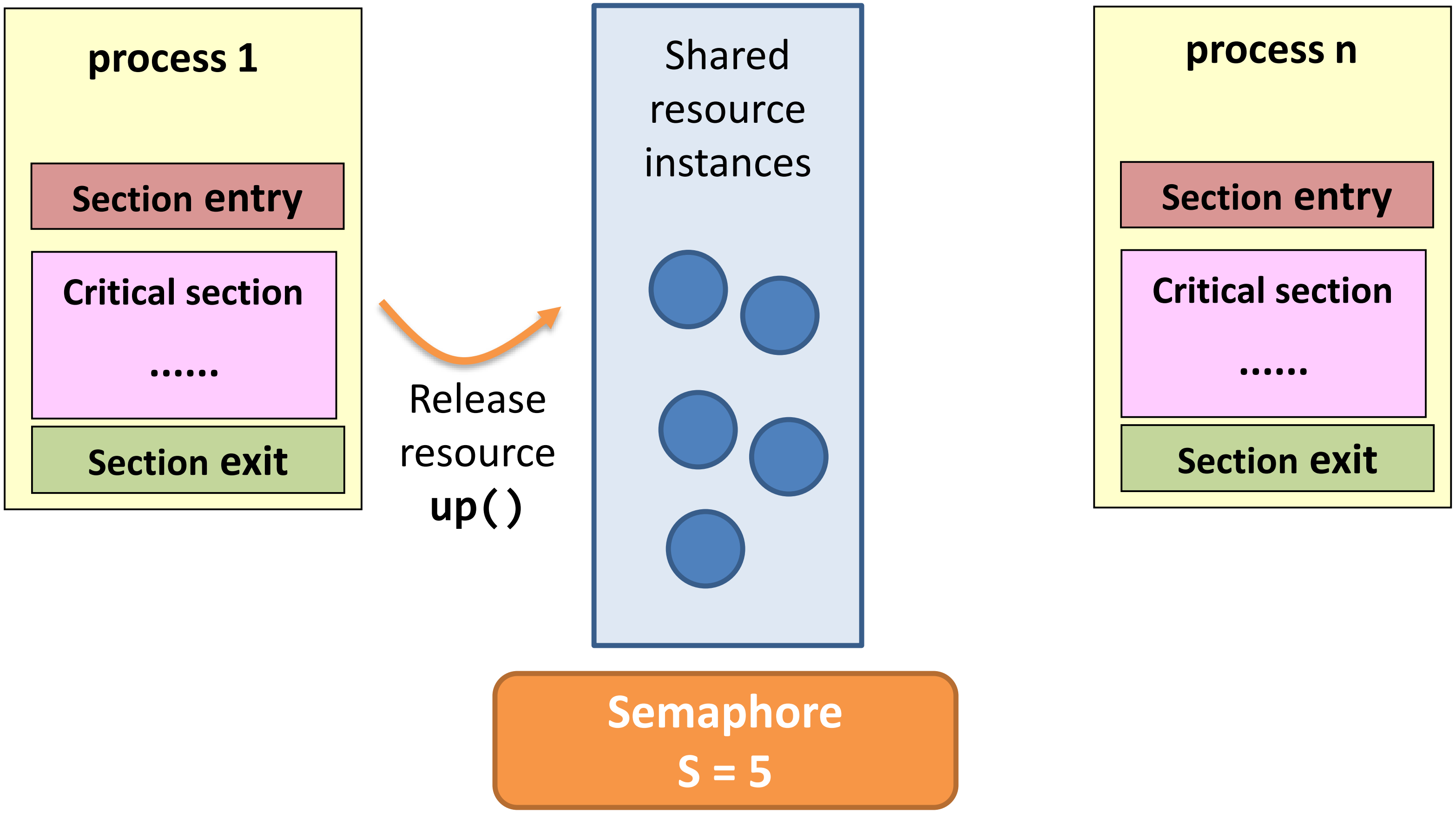
- Idea



Final proposal: Semaphore

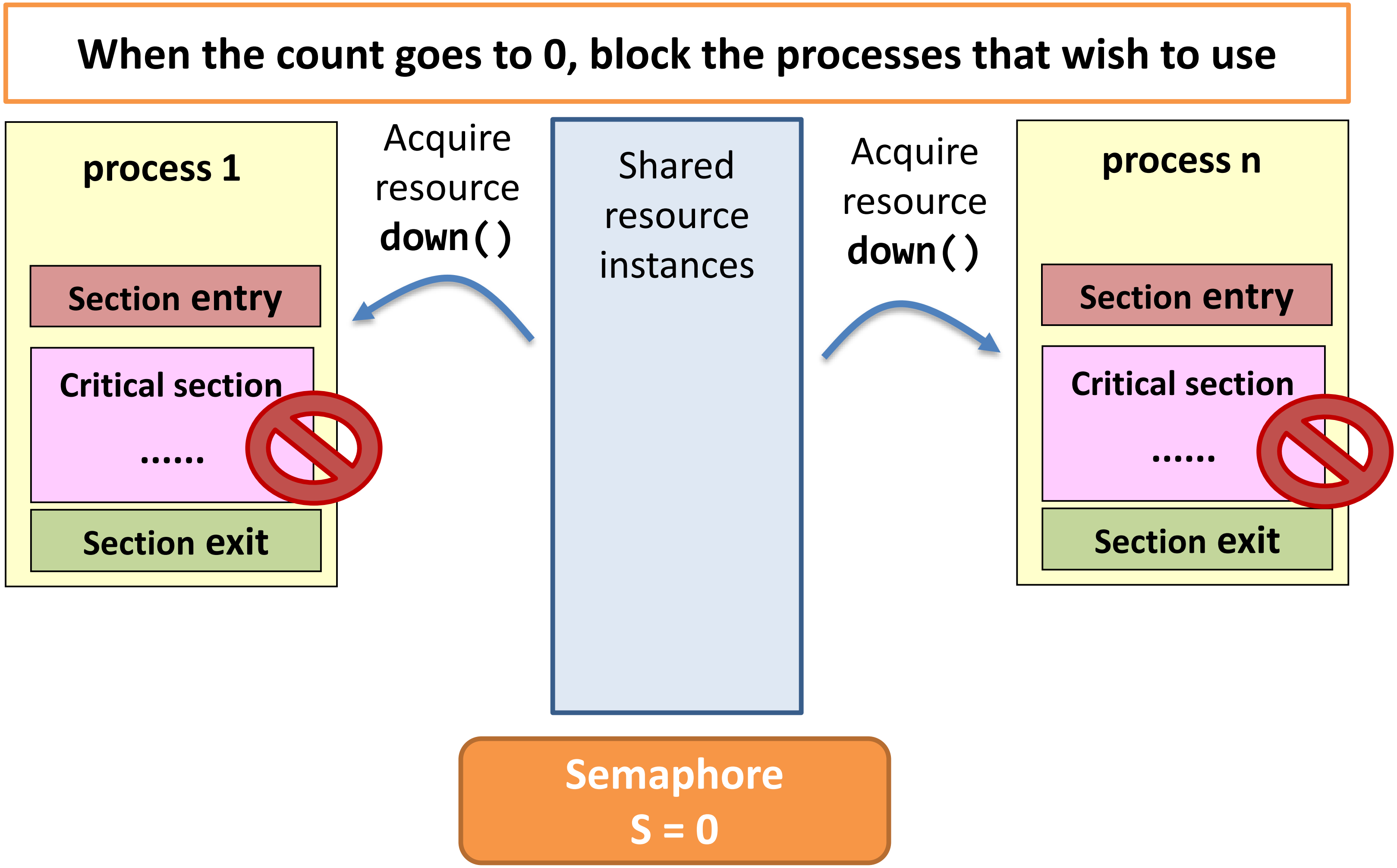
- Idea

Release a resource, perform up() to increment the count



Final proposal: Semaphore

- Idea



Semaphore – Simple Implementation

Data Type definition

```
typedef int semaphore;
```

Counting Semaphore: initialized to be the number of resources available

Section Entry: down()

```
void down(semaphore *s) {
    while ( *s == 0 ) {
            ;//busy waiting
     }
    *s = *s - 1;
}
```

Section Exit: up()

```
void up(semaphore *s) {
    *s = *s + 1;
}
```

Semaphore – Address busy waiting

Data Type definition

```
typedef int semaphore;
```

First issue: Busy waiting

Solution: block the process instead of busy waiting (place the process into a waiting queue)

Section Entry: down()

```
void down(semaphore *s) {
    while ( *s == 0 ) {
        special_sleep();
     }
    *s = *s - 1;
}
```

Section Exit: up()

```
void up(semaphore *s) {
   if ( *s == 0 )
      special_wakeup();
   *s = *s + 1;
}
```

Semaphore – Address busy waiting

Data Type definition

```
typedef int semaphore;
```

```
typedef struct{

     int value;
     struct process * list;

}semaphore;
```

First issue: Busy waiting

Solution: block the process instead of busy waiting (place the process into a waiting queue)

Note

Implementation: The waiting queue may be associated with the semaphore, so a semaphore is not just an integer

Semaphore – Atomicity

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {
    while ( *s == 0 ) {
        special_sleep();
    }
    *s = *s - 1;
}
```

Second issue: Atomicity (both operations must be atomic)

Solution: Disabling interrupts

Section Exit: up()

```
void up(semaphore *s) {
    if ( *s == 0 )
        special_wakeup();
    *s = *s + 1;
}
```

Semaphore – Atomicity

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
    }
    *s = *s - 1;
    enable_interrupt();
}
```

Second issue: Atomicity (both operations must be atomic)

Solution: Disabling interrupts

Also, only one process can invoke "**disable_interrupt()**". Later processes would be blocked until "**enable_interrupt()**" is called.

Section Exit: up()

```
void up(semaphore *s) {
    disable_interrupt();
    if ( *s == 0 )
        special_wakeup();
    *s = *s + 1;
    enable_interrupt();
}
```

Semaphore – The code

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
     }
    *s = *s - 1;
    enable_interrupt();
}
```

Why need these two statements?

Disabling interrupts may sacrifice concurrency, so it is essential to keep the critical section as short as possible

Section Exit: up()

```
void up(semaphore *s) {
    disable_interrupt();
    if ( *s == 0 )
        special_wakeup();
    *s = *s + 1;
    enable_interrupt();
}
```

Semaphore – details

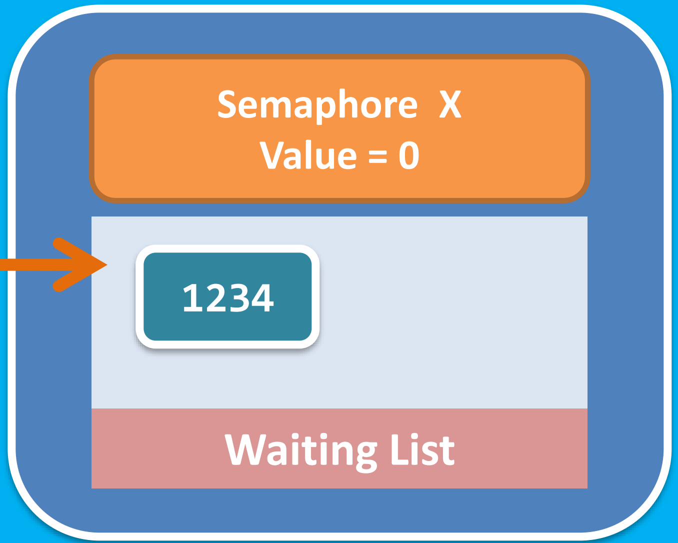
Process 1234

down(X)

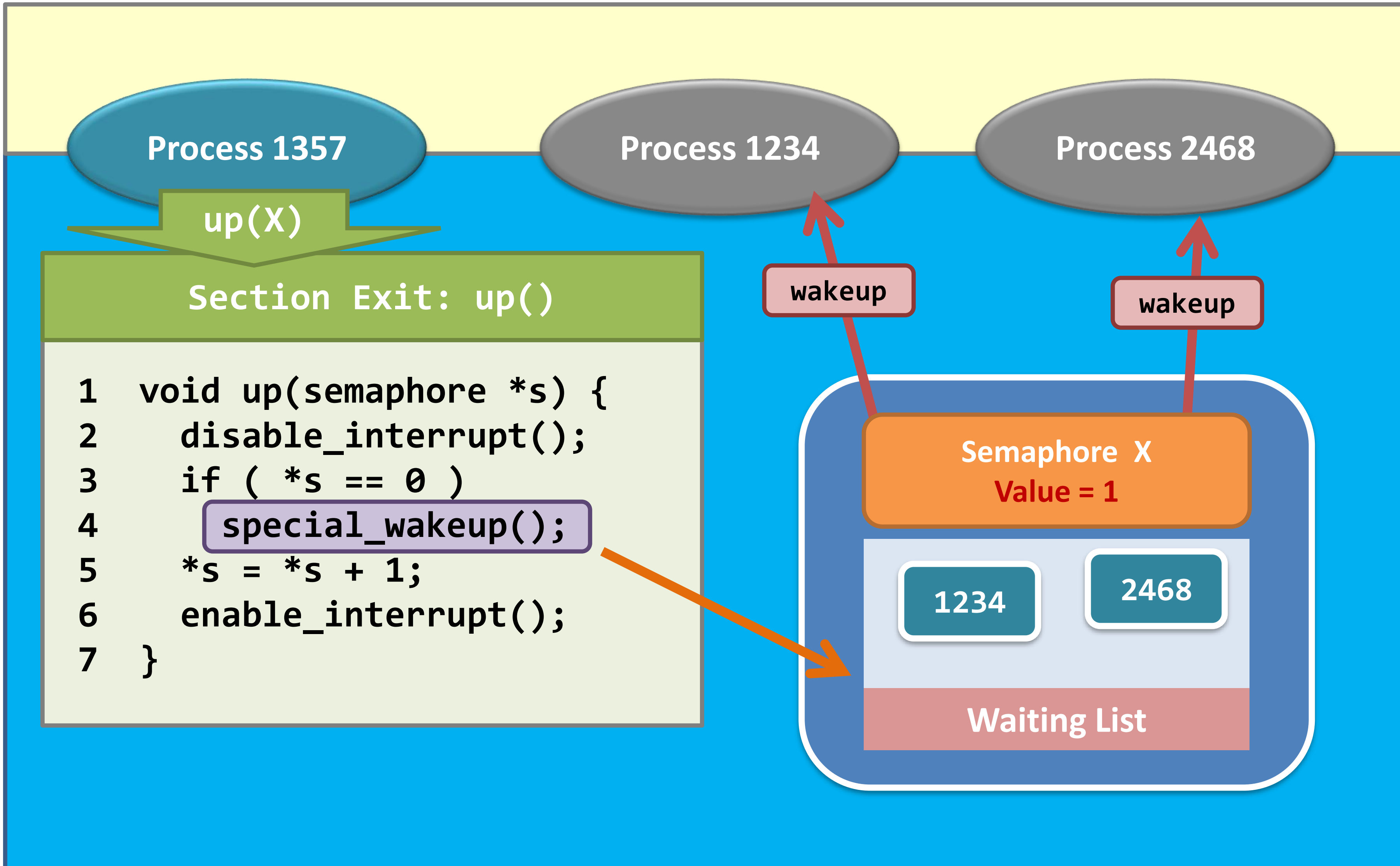
Suppose that process 1234 is willing to access the shared resource (enter its critical section), but no resource is available

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
     }
    *s = *s - 1;
    enable_interrupt();
}
```

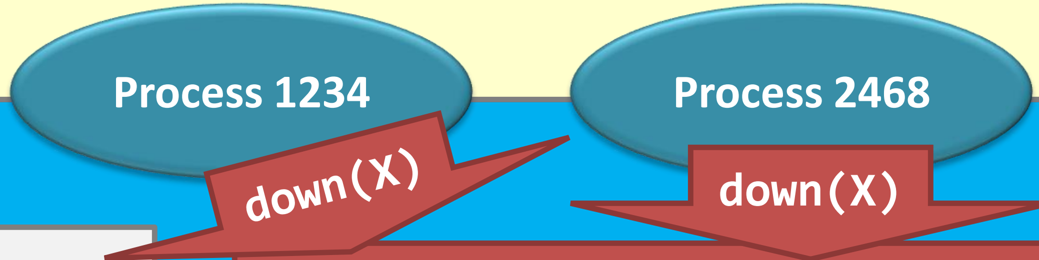


Semaphore – details




```
void up(semaphore *s) {
  disable_interrupt();
  if ( *s == 0 )
    special_wakeup();
  *s = *s + 1;
  enable_interrupt();
}
```

Semaphore – details



Note that it is impossible for **two blocked processes to get out of the down() simultaneously.**

Why?

Only one process can invoke **`disable_interrupt()`**

Only one process can manipulate this shared variable

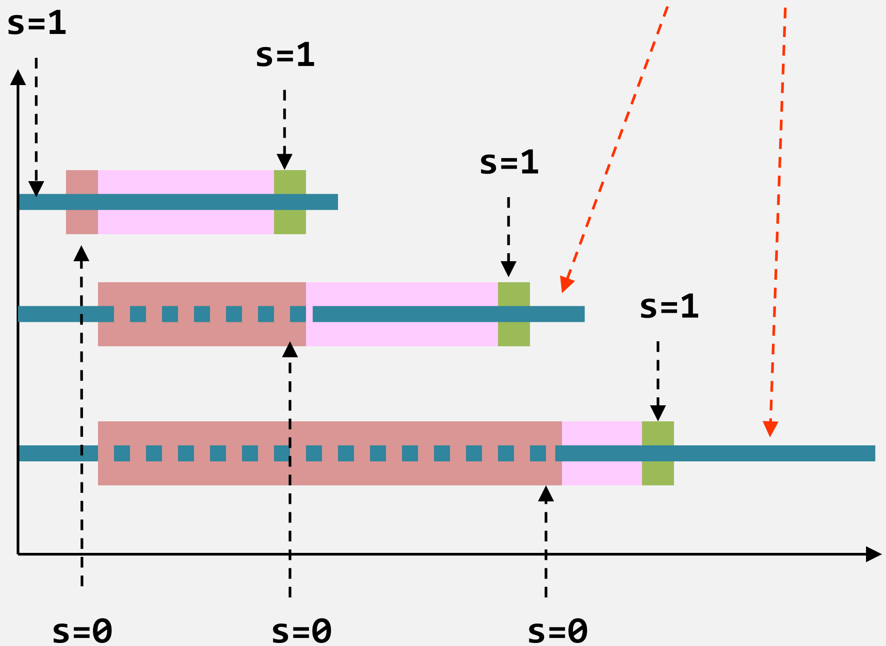
Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();   // ← here
    }
    *s = *s - 1;
    enable_interrupt();
}
```

Semaphore – in action

- Add them together…

Either one of the processes can enter the critical section when the first process calls “**up(s)**”.




```
semaphore *s;
*s = 1;        /* initial value */
```

```
while(TRUE) {
    down(s);
    critical_section();
    up(s);
}
```

entry

exit

Summary…on semaphore

- More on semaphore…it demonstrates an important kind of operations – **atomic operations**.

Definition of atomic operation

- Either none of the instructions of an atomic operation were completed, or
- All instructions of an atomic operation are completed.

- In other words, the entire **up()** and **down()** are indivisible.
  – If it returns, the change must have been made;
  – If it is aborted, no change would be made.

Summary…on critical section problem

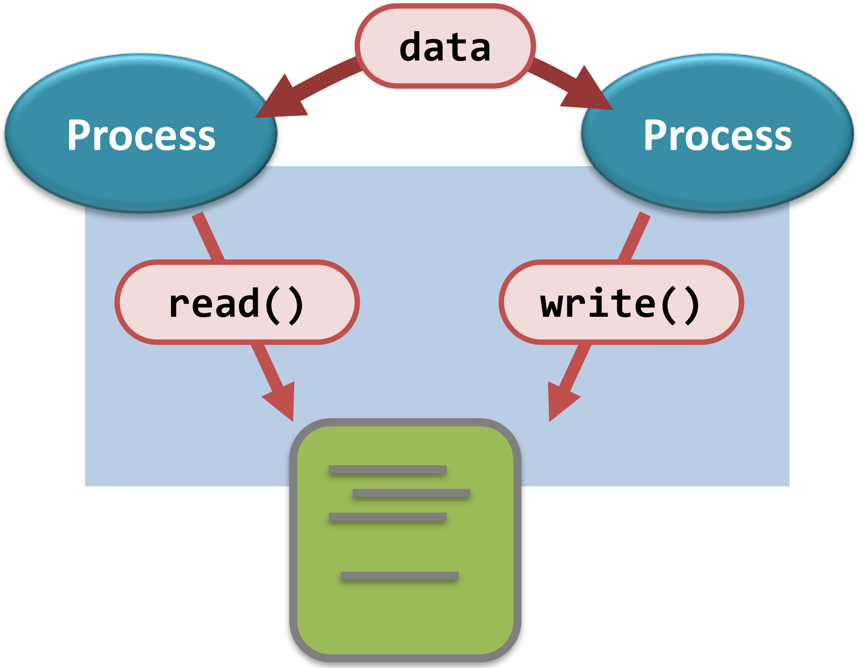
- What happened is just the implementation of mutual exclusion (section entry and section exit).

	Comments
Disabling interrupts	Time consuming for multiprocessor systems, sacrifices concurrency.
Strict alternation	Not a good one, busy waiting & violating one requirement.
Peterson's solution	Busy waiting & has a potential "*priority inversion problem*".
Mutex lock	Busy waiting, often relies on hardware instructions.
Semaphore	**BEST CHOICE**.

- What is next?
  - How to use semaphore to solve classic IPC problems
  - Deadlock

Story so far…

- For shared memory and files, concurrent access may yield unpredictable outcomes
  - **Race condition**



Shared objects

- To avoid race condition, **mutual exclusion** must be guaranteed
  - Critical section
  - Implementations (entry/exit)
    - Hardware instructions
    - Disabling interrupts
    - Strict alternation
    - Peterson's solution
    - Mutex lock
    - **Semaphore**

Semaphore Usage

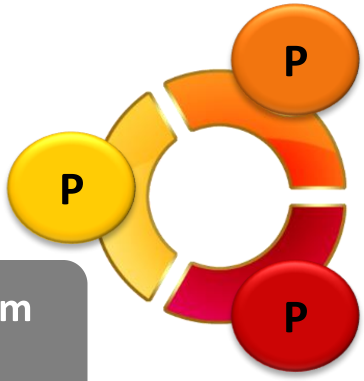
- Semaphore can be used for
  - Mutual exclusion (binary semaphore)
  - Process synchronization (counting semaphore may be needed)

- How to do **process synchronization** w/ semaphore?
  - Mutual exclusion + coordination (multiple semaphores)
  - Careless design may lead to other issues
    - Deadlock

The Deadlock Problem

Classic IPC problems

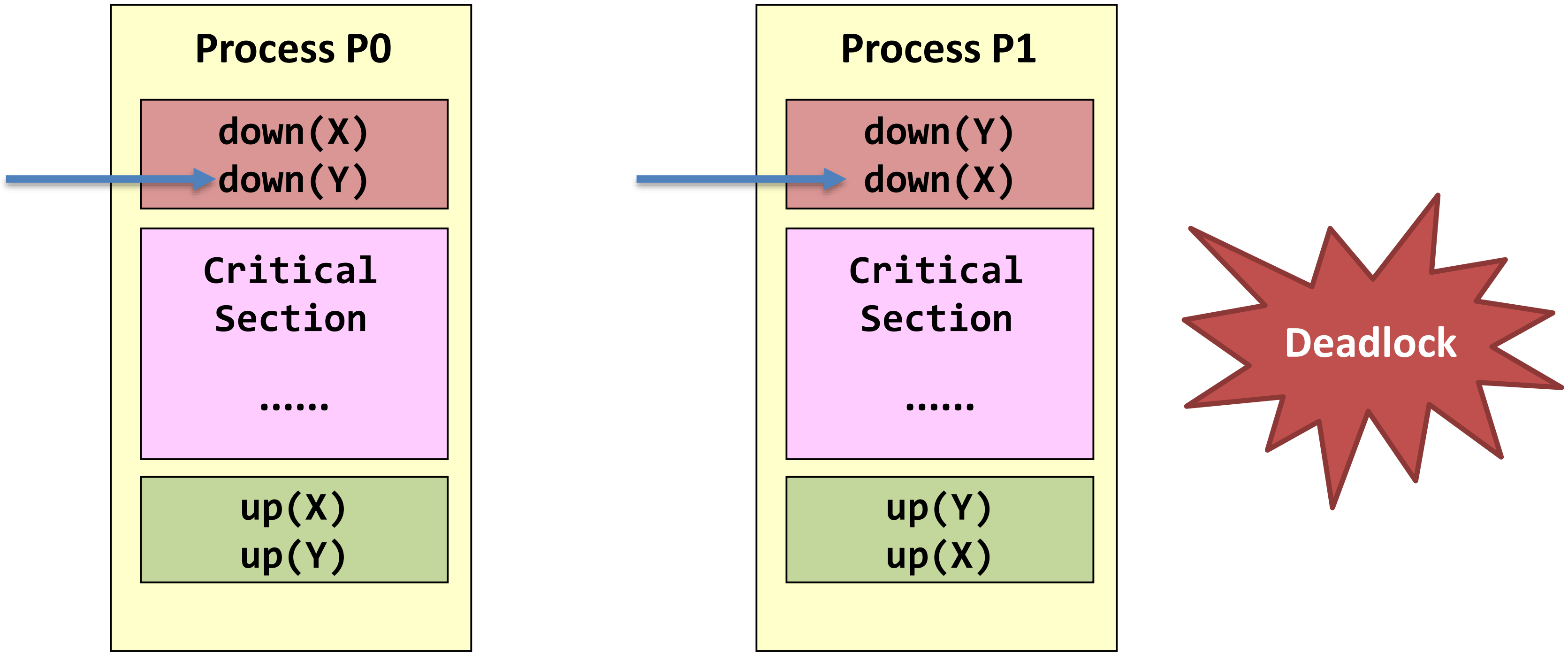
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem



Let's teach them not to fight.

Deadlock Example

- Problems when using semaphore



Scenario: P0 must wait until P1 executes **up(Y)**, P1 must wait until P0 executes **up(X)**

Deadlock Requirements

- **<u>Requirement #1: Mutual Exclusion</u>**.
  – Only one process at a time can use a resource

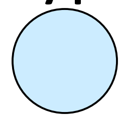
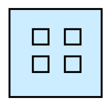
- **<u>Requirement #2. Hold and wait</u>.**
  – A process must be holding at least one resource and waiting to acquire additional resources held by other processes

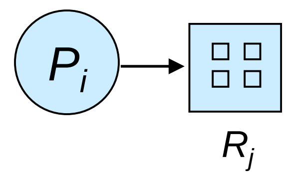
Deadlock Requirements

- **Requirement #3: No preemption.**
  - A resource can be released only voluntarily by the process holding it after that process has completed its task

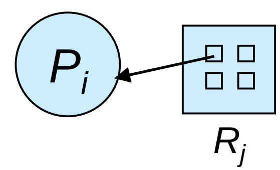
- **Requirement #4. Circular wait.**
  - There exists a set {P_0, P_1, …, P_n} of waiting processes such that P_0 waits for P_1, P_1 waits for P_2, …, P_{n-1} waits for P_n , P_n waits for P_0

How to Handle Deadlocks

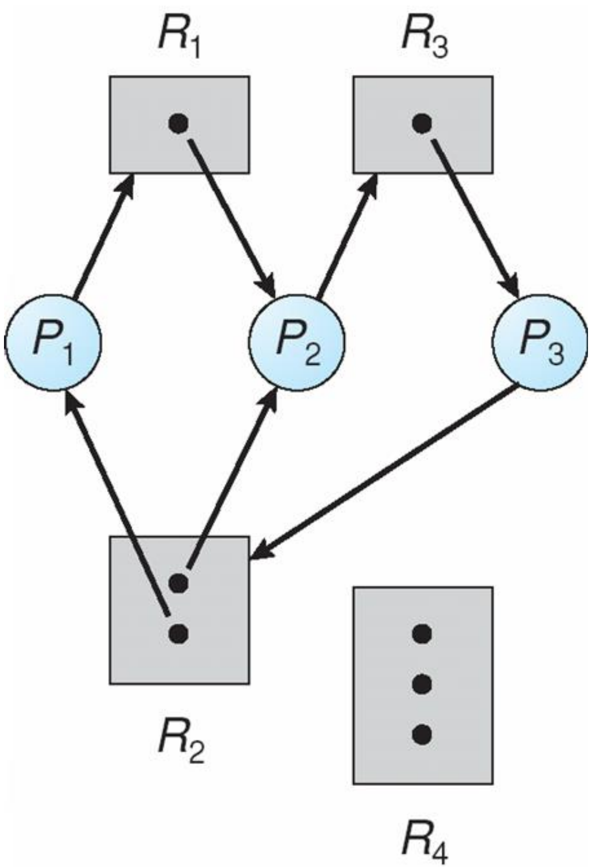
- Deadlock characterization: Deadlocks can be described using **resource-allocation graph**
  - Set V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$: processes 
    - $R = \{R_1, R_2, \ldots, R_m\}$: all resource types (each type may have multiple instances) 
  - Set E
    - **request edge** – directed edge $P_i \rightarrow R_j$

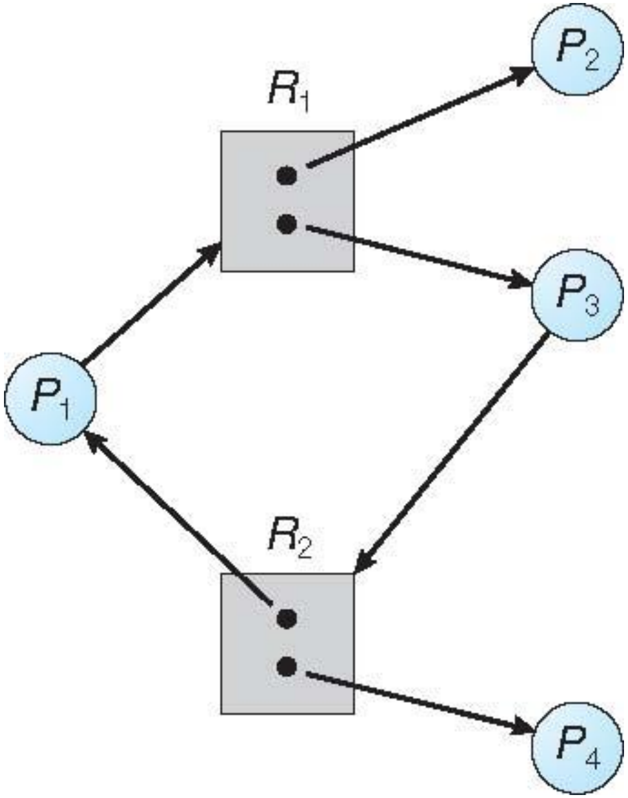


    - **assignment edge** – directed edge $R_j \rightarrow P_i$



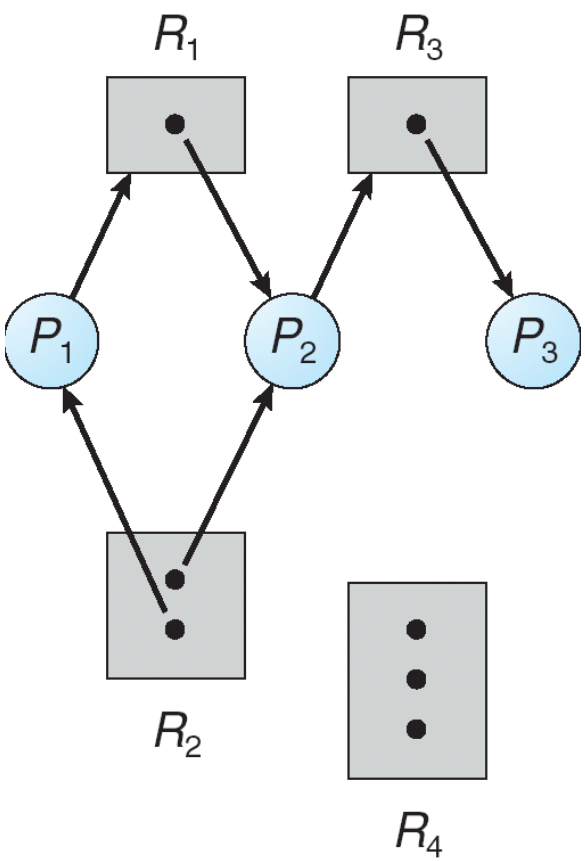
Examples





How to Handle Deadlocks

- **Detect** deadlock and recover
  - Resource-allocation graph: detect the existence of a cycle

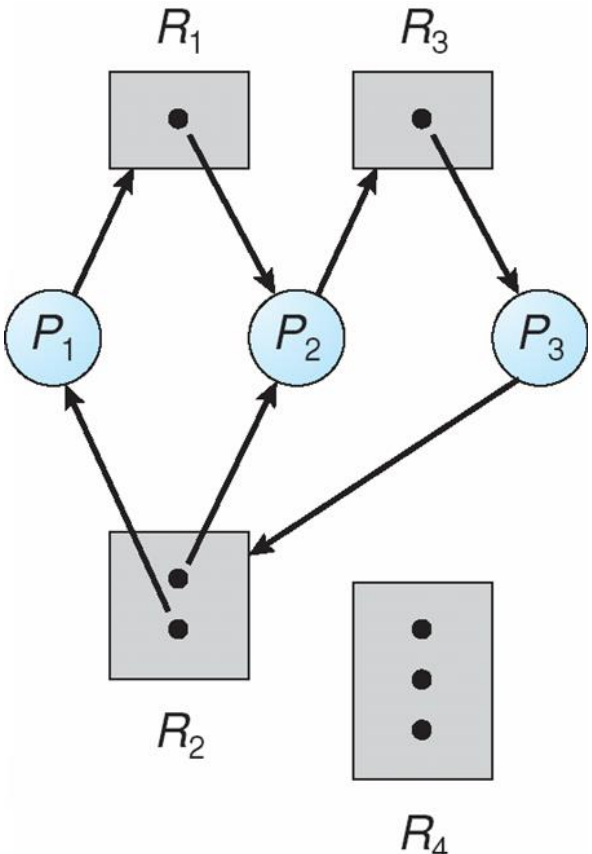


No cycles
No deadlock

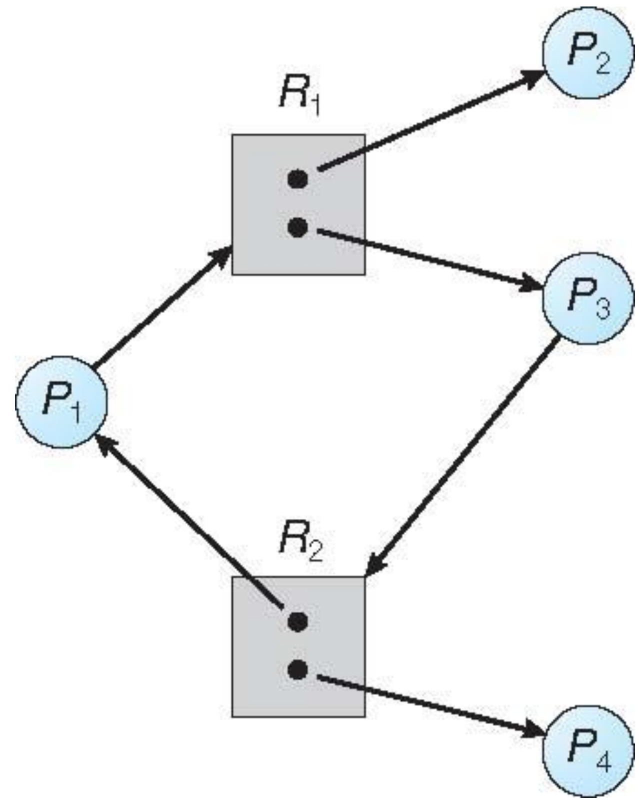
Contains a cycle
Case 1: only one instance per resource type: deadlock

Examples

- **Detect** deadlock and recover
  - What if each resource has multiple instances



Deadlock



No deadlock

How to Handle Deadlocks

- **Detect** deadlock and recover
  - What if each resource has multiple instances
    - Matrix method: four data structures
      - Existing (total) resources (m types): $(E_1, E_2, \dots, E_m)$
      - Available resources: $(A_1, A_2, \dots, A_m)$
      - Allocation matrix: $\begin{bmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{bmatrix}$ (C_{ij}: # of type-j resources held by process i)
      - Request matrix: $\begin{bmatrix} R_{11} & \cdots & R_{1m} \\ \vdots & \ddots & \vdots \\ R_{n1} & \cdots & R_{nm} \end{bmatrix}$ (R_{ij}: # of type-j resources requested by process i)

➢ Repeatedly check P_i s.t. $\boldsymbol{R}_i \le \boldsymbol{A}$? ($P_i$ can be satisfied?)
  - ✓ Yes: $\boldsymbol{A} = \boldsymbol{A} + \boldsymbol{C}_i$ (release resources)
  - ✓ No: End (remaining processes are deadlocked)

How to Handle Deadlocks

- **Prevent/avoid** deadlocks: Banker's algorithm
  - Idea: check system state defined by (E, A, C, R)
    - **Safe state**: exist one running sequence to guarantee that all processes' demand can be satisfied

Existing resources

Maximum demand

Processes

A	3	9
B	2	4
C	2	7

Available: 3

A	3	9
B	4	4
C	2	7

Available: 1

A	3	9
B	0	-
C	2	7

Available: 5

A	3	9
B	2	4
C	7	7

Available: 0

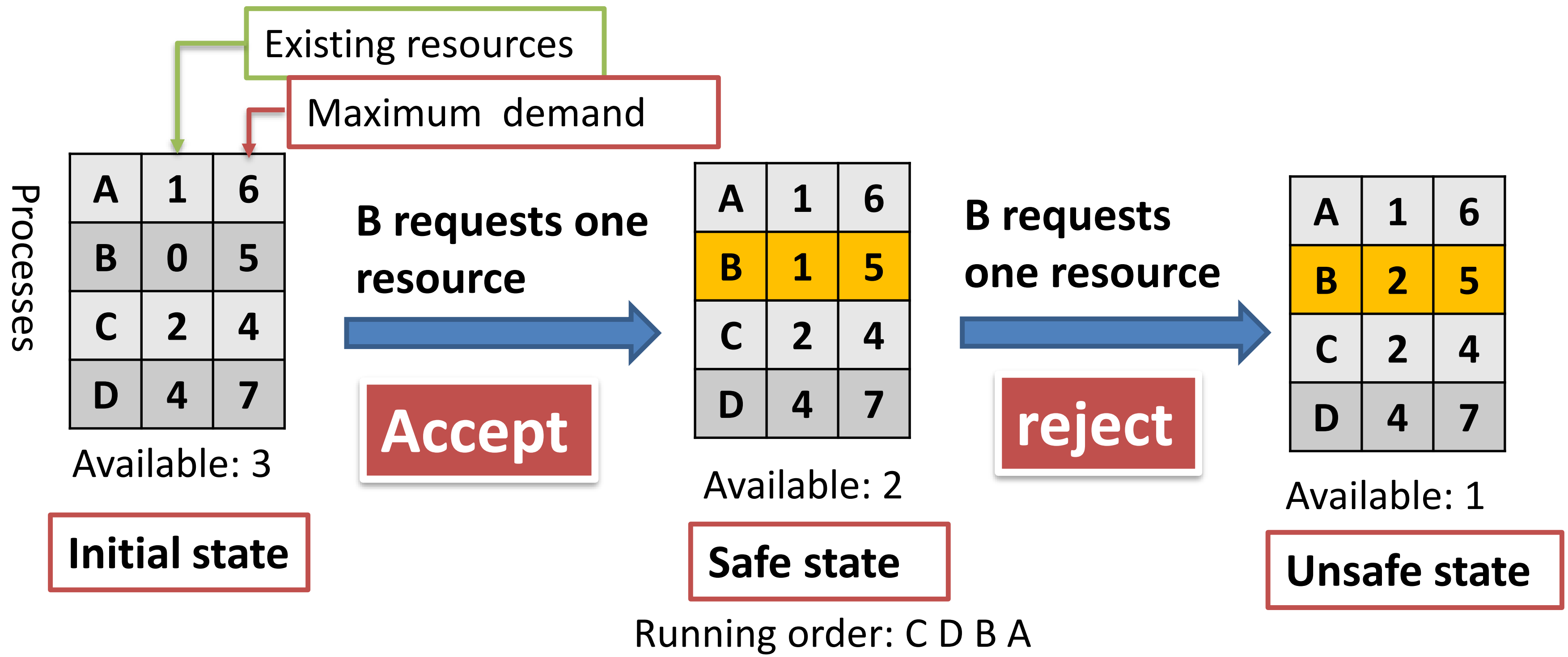
A	3	9
B	2	4
C	0	-

Available: 7

    - **Unsafe state**: Not exist any sequence to guarantee the demand
      – It is not deadlock (it can still run for some time/processes may release some resources)

How to Handle Deadlocks

- **Prevent/avoid** deadlocks: Banker's algorithm
  - For each request: safe (accept), unsafe (reject)



The algorithm can also be extended to the case of multiple resources, but it needs to know the demand

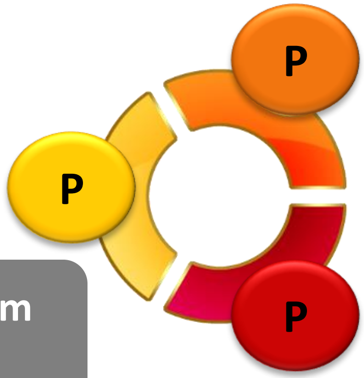
How to Handle Deadlocks

- **Ignore** the problem and pretend that deadlocks never occur (stop functioning and restart manually)
  - 鸵鸟算法（假装没发生）
  - Used by most operating systems, including UNIX and windows
  - Deadlocks occur infrequently, avoiding/detecting it is expensive
- A deadlock-free solution does not eliminate **starvation**

The Deadlock Problem

Classic IPC problems

- Dining philosopher problem
- Producer-consumer problem
- Reader-writer problem



Let's teach them not to fight.

What are the problems?

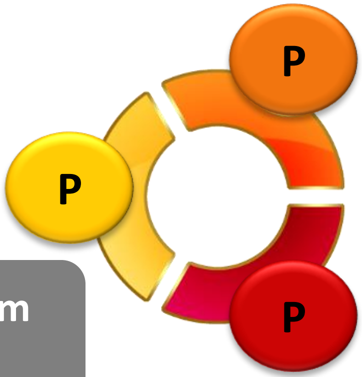
- All the IPC classical problems use **semaphores** to fulfill the synchronization requirements.

	Properties	Examples
Producer-Consumer Problem	Two classes of processes: **producer** and **consumer**; At least one producer and one consumer.	FIFO buffer, such as pipe.
Dining Philosophy Problem	They are all running the same program; At least two processes.	Cross-road traffic control.
Reader-Writer Problem	Two classes of processes: **reader** and **writer**. No limit on the number of the processes of each class.	Database.

The Deadlock Problem

Classic IPC problems

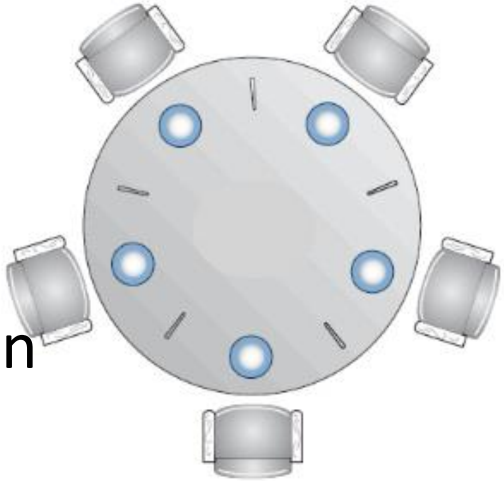
- **Dining philosopher problem**
- Producer-consumer problem
- Reader-writer problem



Let’s teach them not to fight.

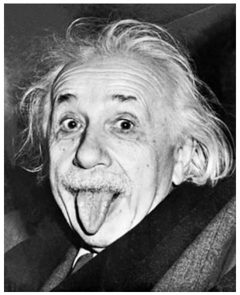
Dining philosopher – introduction

- 5 philosophers, 5 plates of spaghetti, and 5 chopsticks.



- The jobs of each philosopher are
  - to think and
  - to eat: They **need exactly two chopsticks** in order to eat the spaghetti.
- Question: how to construct a synchronization protocol such that
  - they will not result in any **deadlocking scenarios**, and
  - they will not be **starved to death**

Dining philosopher – introduction



Philosophers → Process

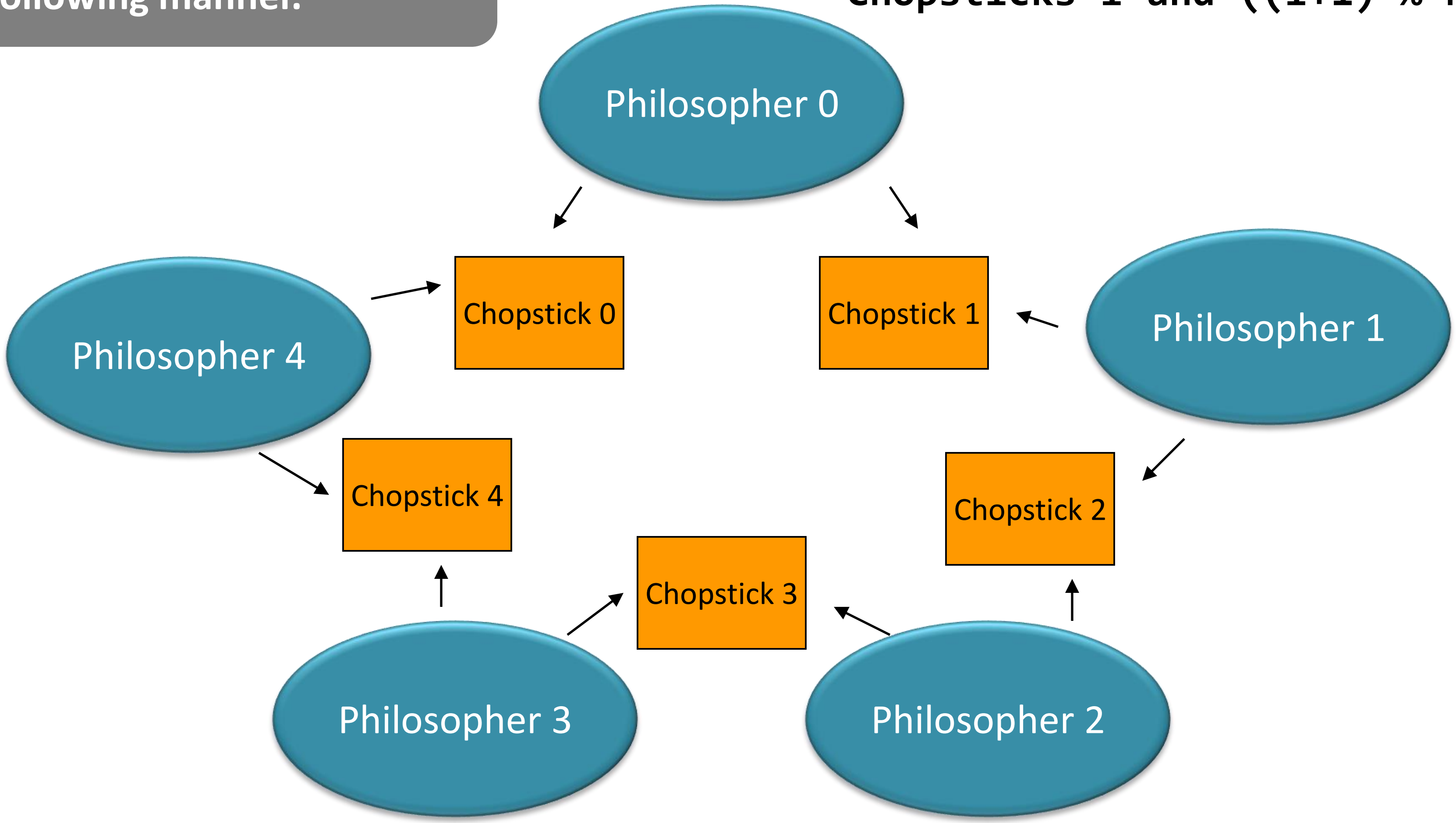
Chopsticks → Shared Object

Spaghetti → Consider to have infinite supply.

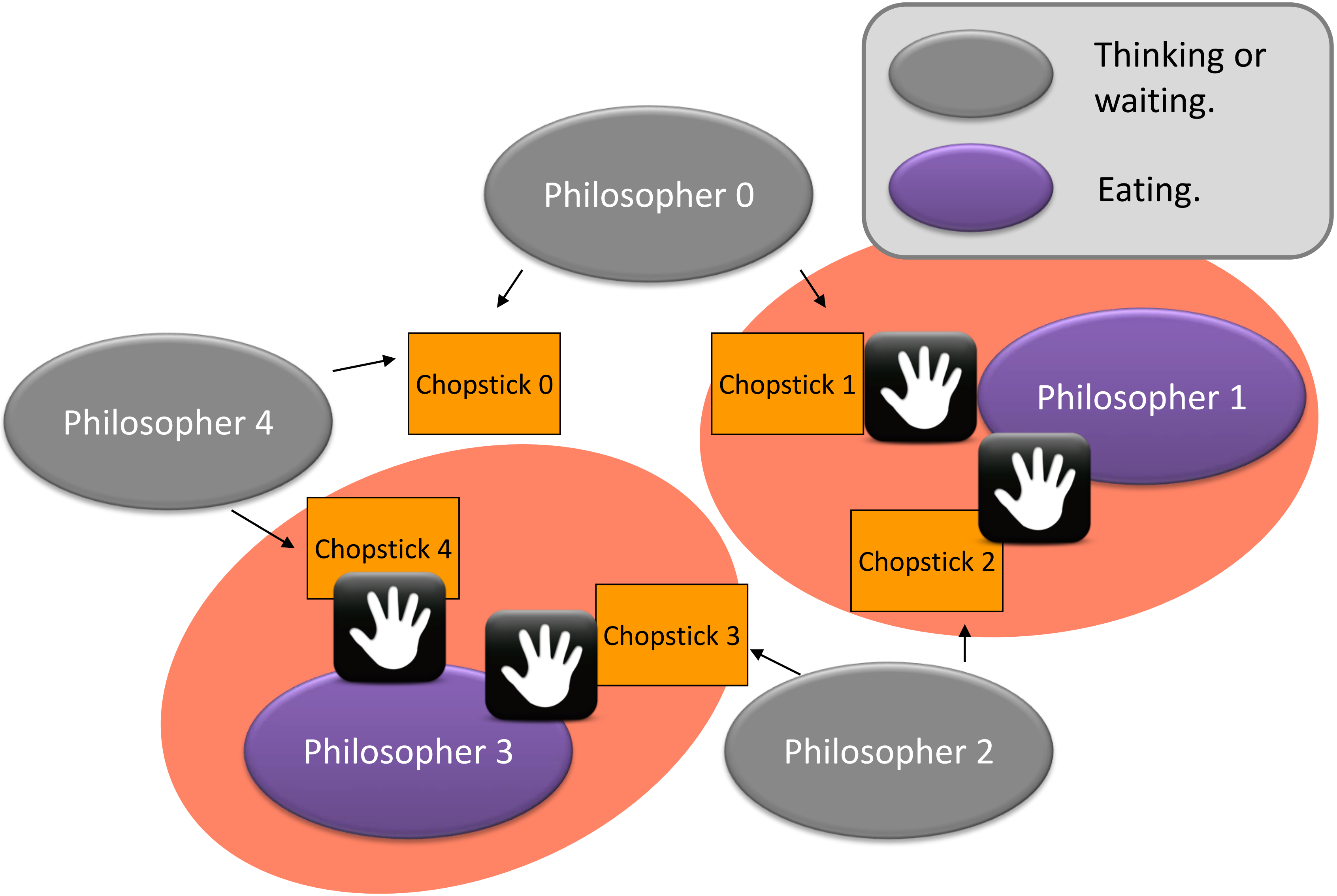
Dining philosopher – introduction

The chopsticks are arranged in the following manner.

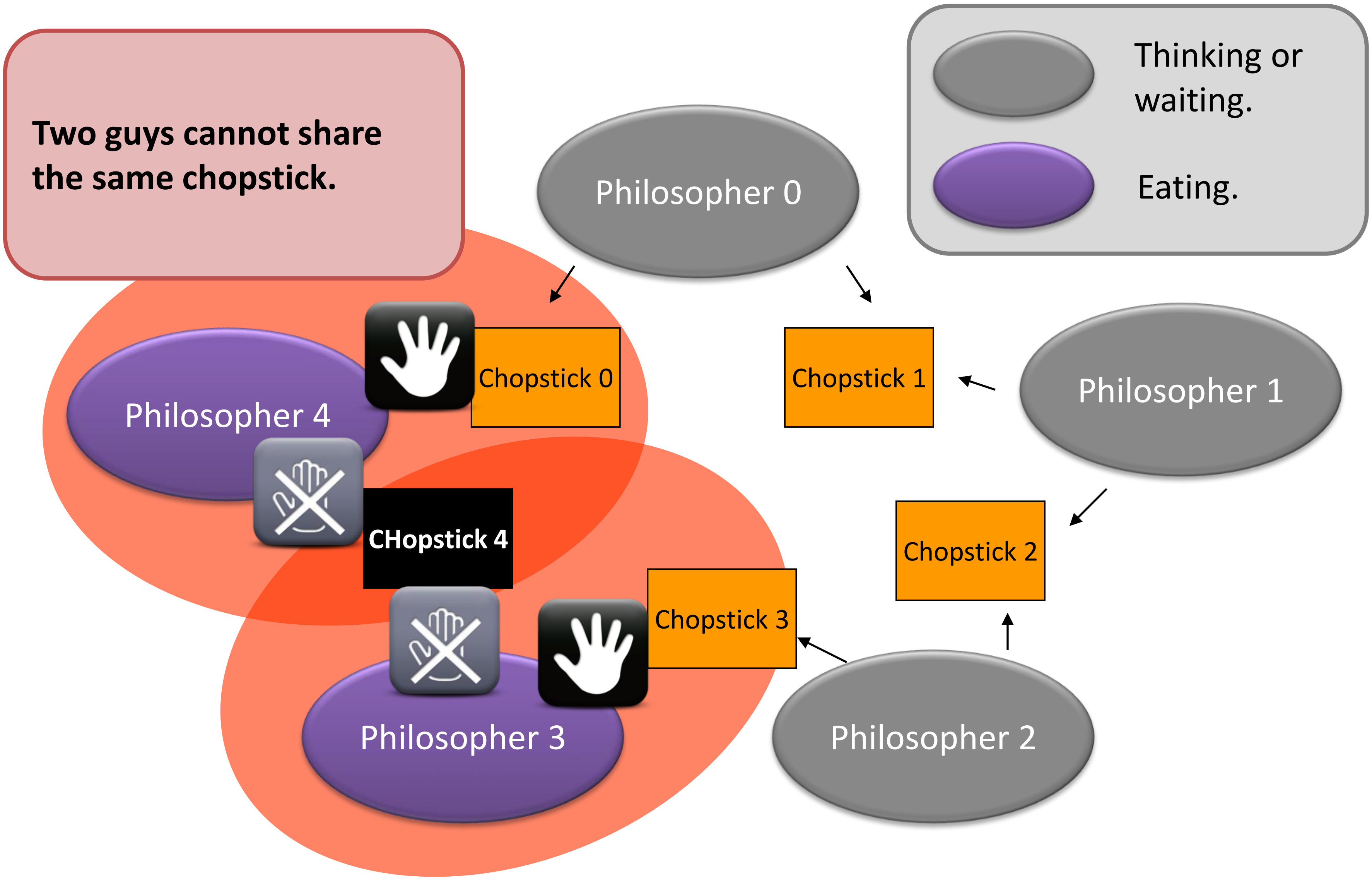
```
Philosopher i needs
Chopsticks i and ((i+1) % N);
```



Dining philosopher – introduction



Dining philosopher – introduction



Dining philosopher – requirement #1

- <u>**Mutual exclusion**</u>
  - What if there is no mutual exclusion?
    - Then: while you're eating, the two men besides you will and must **steal all your chopsticks!**

- Let's proposal the following solution:
  - When you are hungry, you have to check if anyone is using the chopstick that you need.
  - If yes, you have to wait.
  - If no, **seize both chopsticks.**
  - After eating, put down all your chopsticks.

Dining philosopher – meeting requirement #1?

A quick question: what should be initial values?

Shared object

```
#define  N  5
semaphore chop[N];
```

Helper Functions

```
void take(int i) {
    down(&chop[i]);
}

void put(int i) {
    up(&chop[i]);
}
```

Section Entry

Critical Section

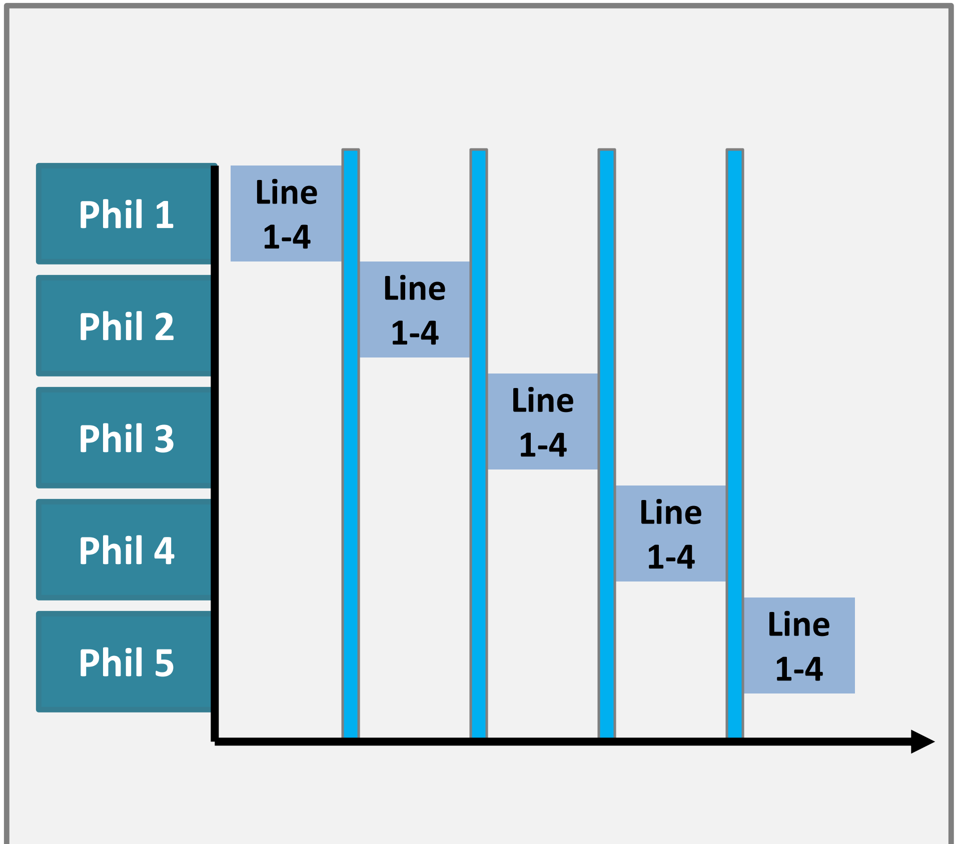
Section Exit

Main Function

```
void philosopher(int i) {
    while (TRUE) {
        think();

        take(i);
        take((i+1) % N);

        eat();

        put(i);
        put((i+1) % N);
    }
}
```

Dining philosopher – meeting requirement #1?

Final Destination: Deadlock!



Main Function

```
1  void philosopher(int i) {
2      while (TRUE) {
3          think();

4          take(i);
5          take((i+1) % N);

6          eat();

7          put(i);
8          put((i+1) % N);
9      }
10 }
```

Dining philosopher – requirement #2

- **Synchronization**
  - Should avoid any **potential deadlocking execution order**.

- How about the following suggestions:
  - First, a philosopher **takes a chopstick**.
  - If a philosopher finds that he cannot take the second one, then he should **put down the first chopstick**.
  - Then, the philosopher **goes to sleep** for a while.
  - Again, the philosopher tries to get both chopsticks until both ones are seized.

Dining philosopher – meeting requirement #2?

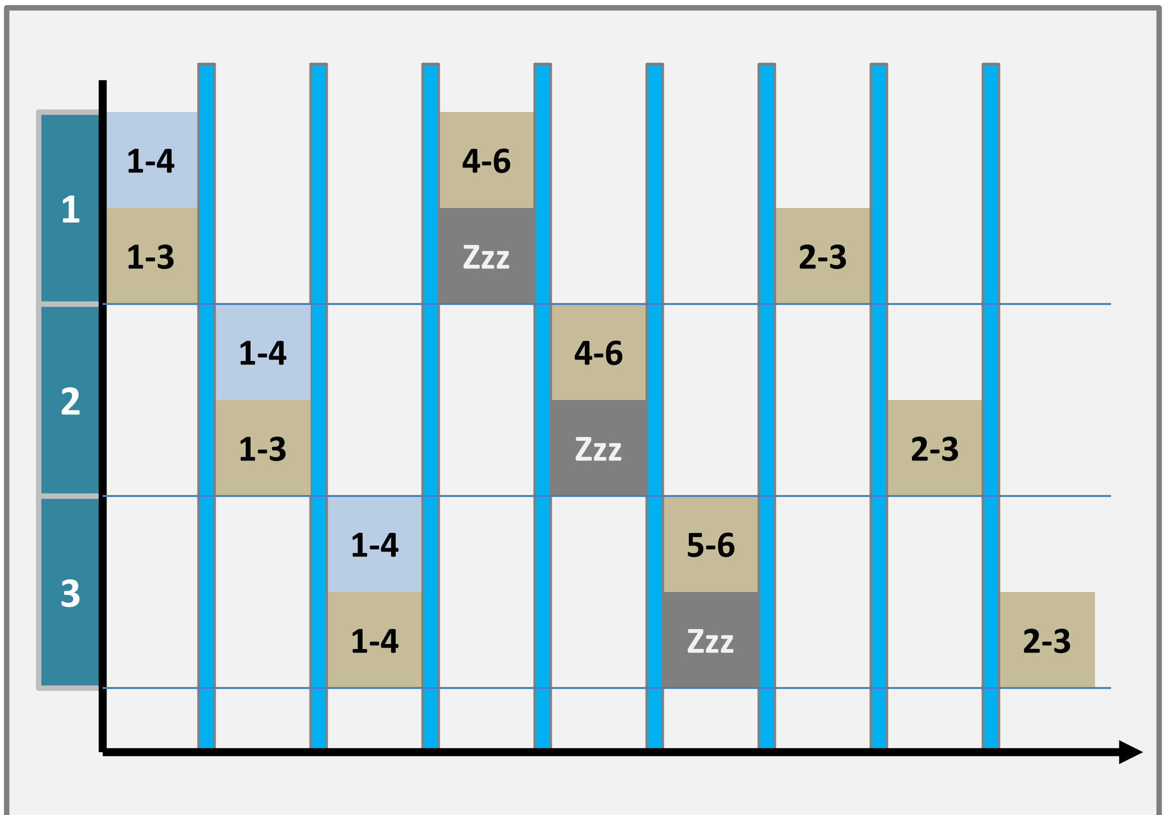
The code: meeting requirement #2?

```
void philosopher(int i) {
    while (TRUE) {
      think();
      take(i);
      eat();
      up(&chop[i]);
      up(&chop[(i+1)%N]);
    }
}
```

```
void take(int i) {
  while(TRUE) {
    down(&chop[i]);
    if (isUsed((i+1)%N)) {
      up(&chop[i]);
      sleep(1);
    }
    else {
      down(&chop[(i+1)%N]);
      break;
    }
  }
}
```

Dining philosopher – meeting requirement #2?

Potential Problem: Philosophers are all busy but no progress were made!



Assume N = 3 (because the space is limited)

```
void take(int i) {
  while(TRUE) {
    down(&chop[i]);
    if (isUsed((i+1)%N)) {
      up(&chop[i]);
      sleep(1);
    }
    else {
      down(&chop[(i+1)%N]);
      break;
    }
  }
}
```

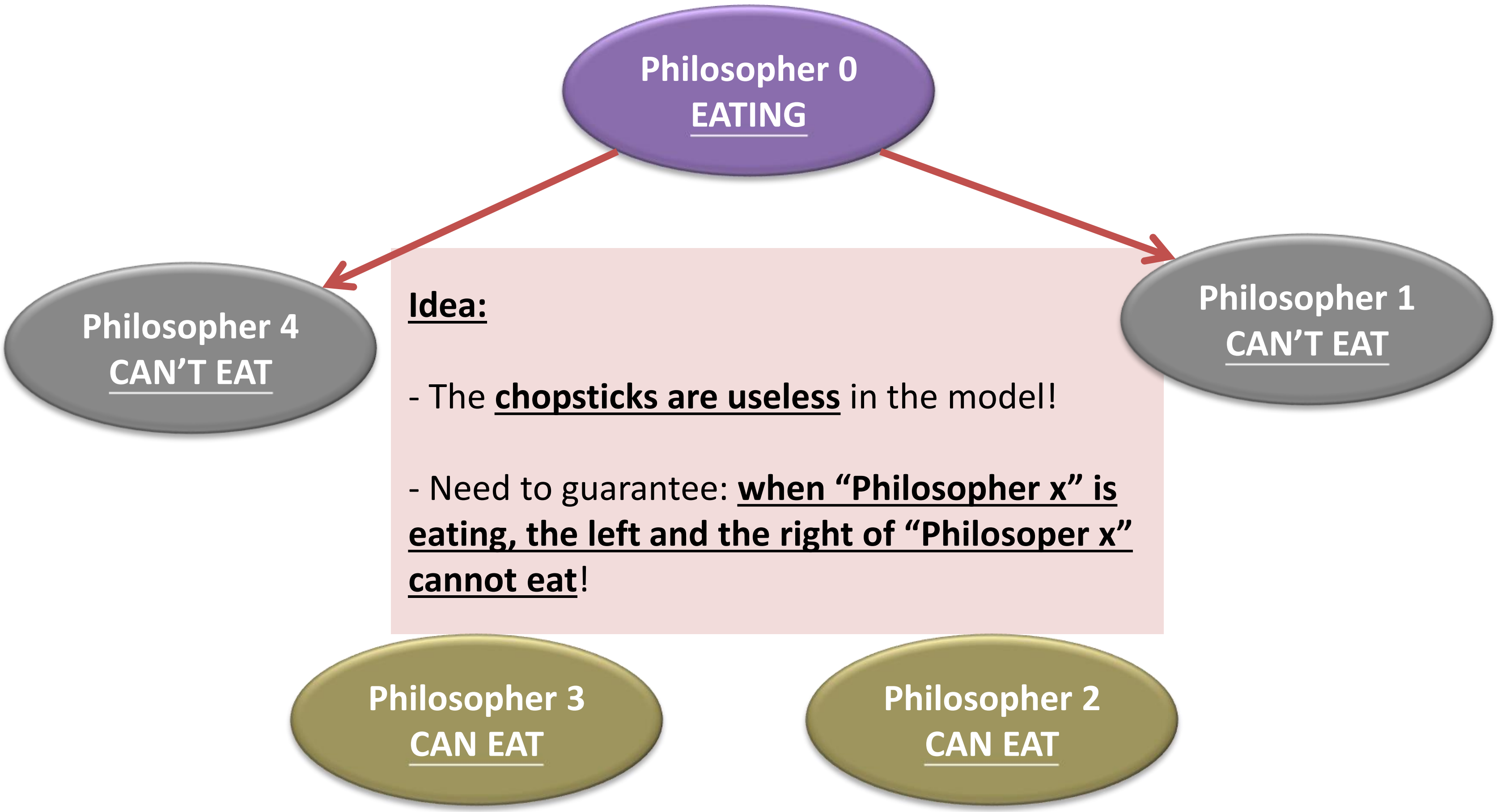
```
void philosopher(int i) {
    while (TRUE) {
      think();
      take(i);
      eat();
      up(&chop[i]);
      up(&chop[(i+1)%N)]);
    }
}
```

Dining philosopher – before the final solution.

- Before we present the final solution, let's see what are the problems that we have.

Problems
<u>Model a chopstick as a semaphore is intuitive, but is not working.</u> The problem is that we are afraid to "**`down()`**", as that may lead to a deadlock.
<u>Using `sleep()` to avoid deadlock is effective, yet bringing another problem.</u> We can always create an execution order that keeps all the philosophers busy, but without useful output.

Dining philosopher – before the final solution.



Idea:

- The **chopsticks are useless** in the model!

- Need to guarantee: **when "Philosopher x" is eating, the left and the right of "Philosoper x" cannot eat**!

Dining philosopher – the final solution.

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Main function

```
void philosopher(int i) {
    think();
    take(i);
    eat();
    put(i);
}
```

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Section exit

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

I will explain the code later.

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Dining philosopher – the final solution.

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Going "left" and "right" in a circular manner.

The states of the philosophers, including "**EATING**", "**THINKING**", and "**HUNGRY**".

Remember, this is shared array.

To guarantee mutual exclusive access to the "**state[N]**" array.

<u>Guess:</u>

What is the meaning of the semaphore s[N]?

To fulfill the synchronization requirement.

<u>Question.</u> What are the initial values of the "**s[N]**" array?

Dining philosopher – the final solution.

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Question. What are they doing?

If both chopsticks are available, I eat. Else, I sleep.

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

If they are eating, I can't be eating.

Dining philosopher – the final solution.

Try to let the one on the **left of the caller** to eat.

Try to let the one on the **right of the caller** to eat.

Section exit

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Wake up the one who can eat!

Dining philosopher – the final solution.

An illustration: How can Philosopher 1 start eating?

Philosopher 0
THINKING

Philosopher 4
THINKING

Note: no chopsticks objects will be shown in this illustration because we don't need them now.

Philosopher 1
THINKING

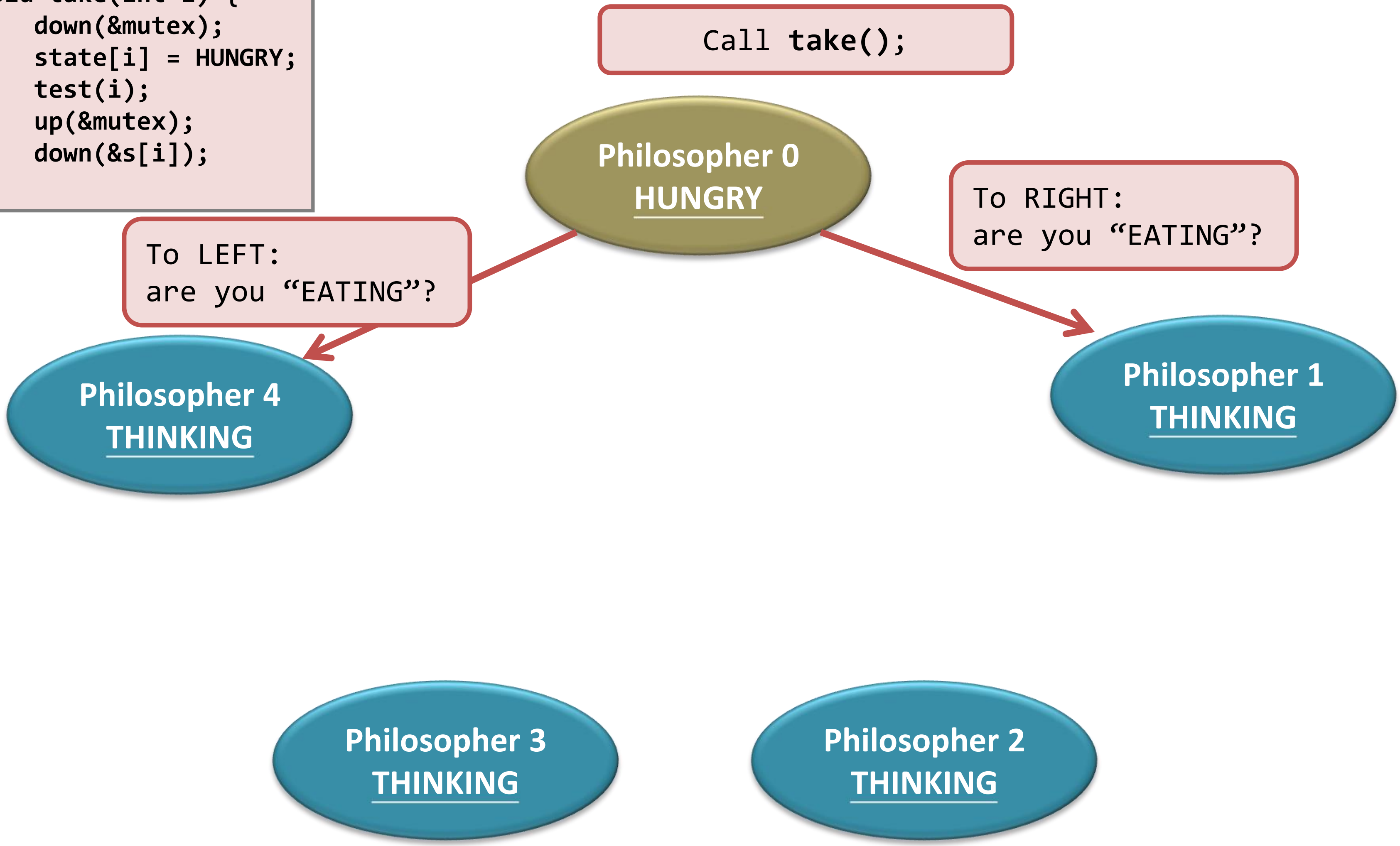
Philosopher 3
THINKING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

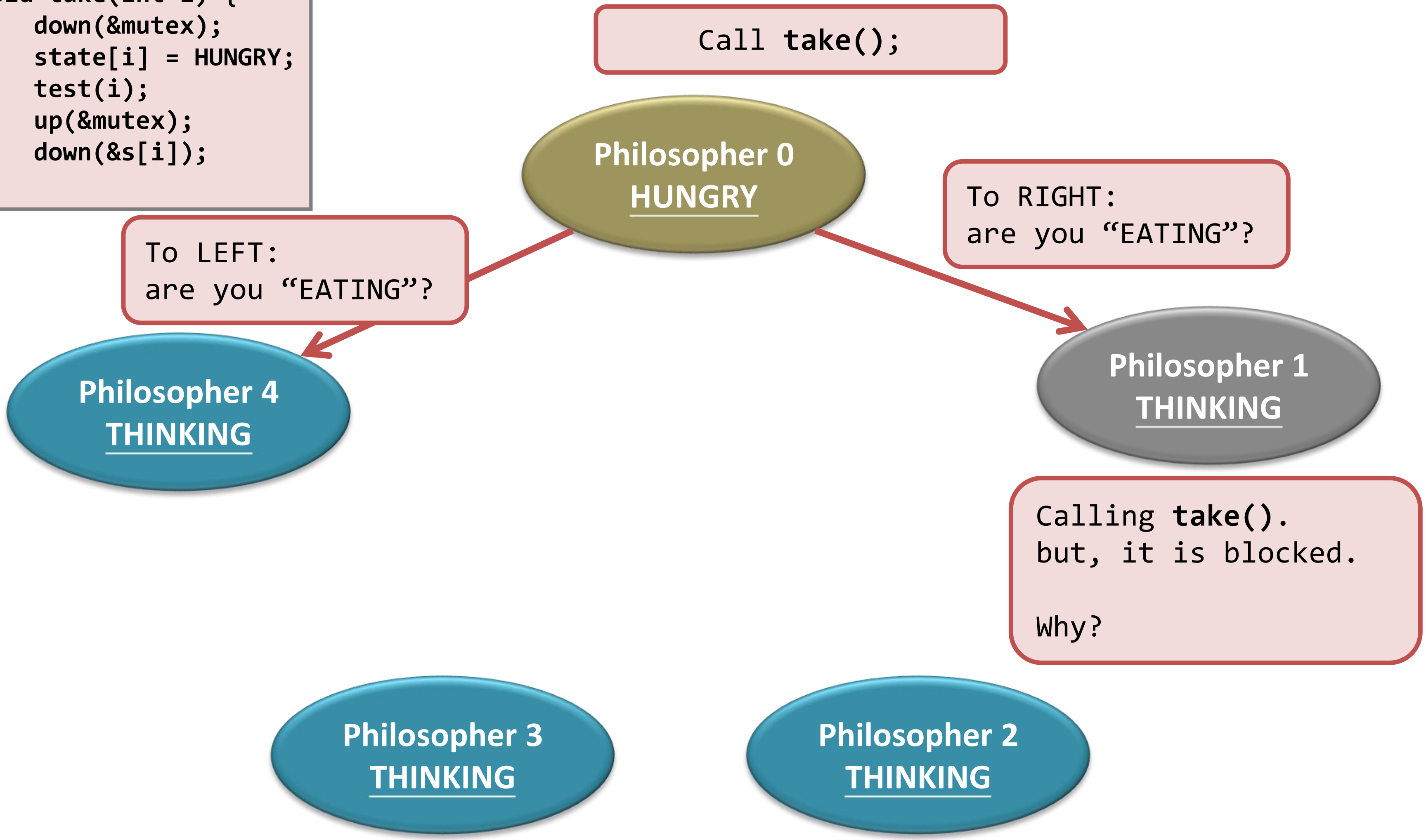
```
void take(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

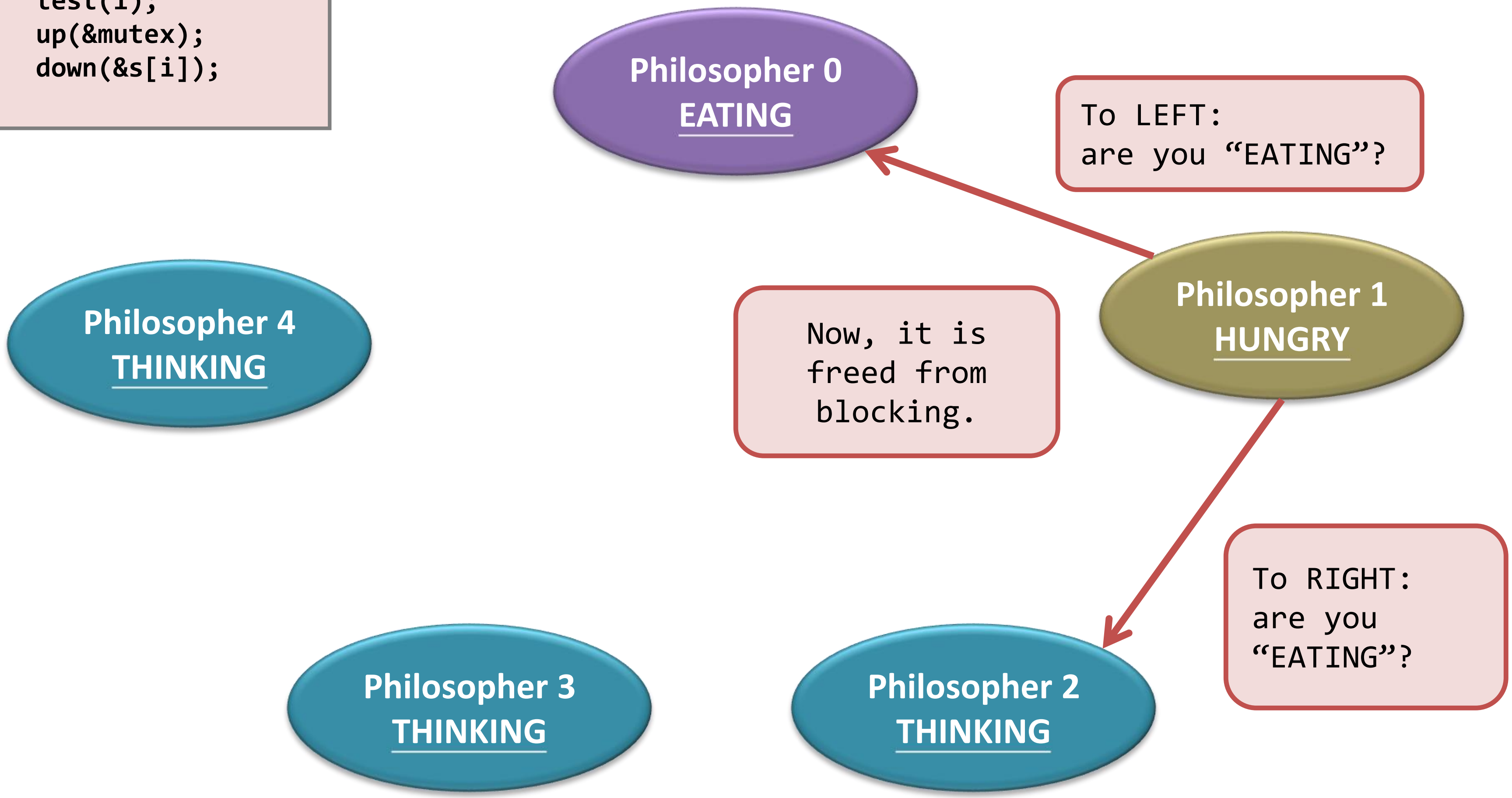
```
void take(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

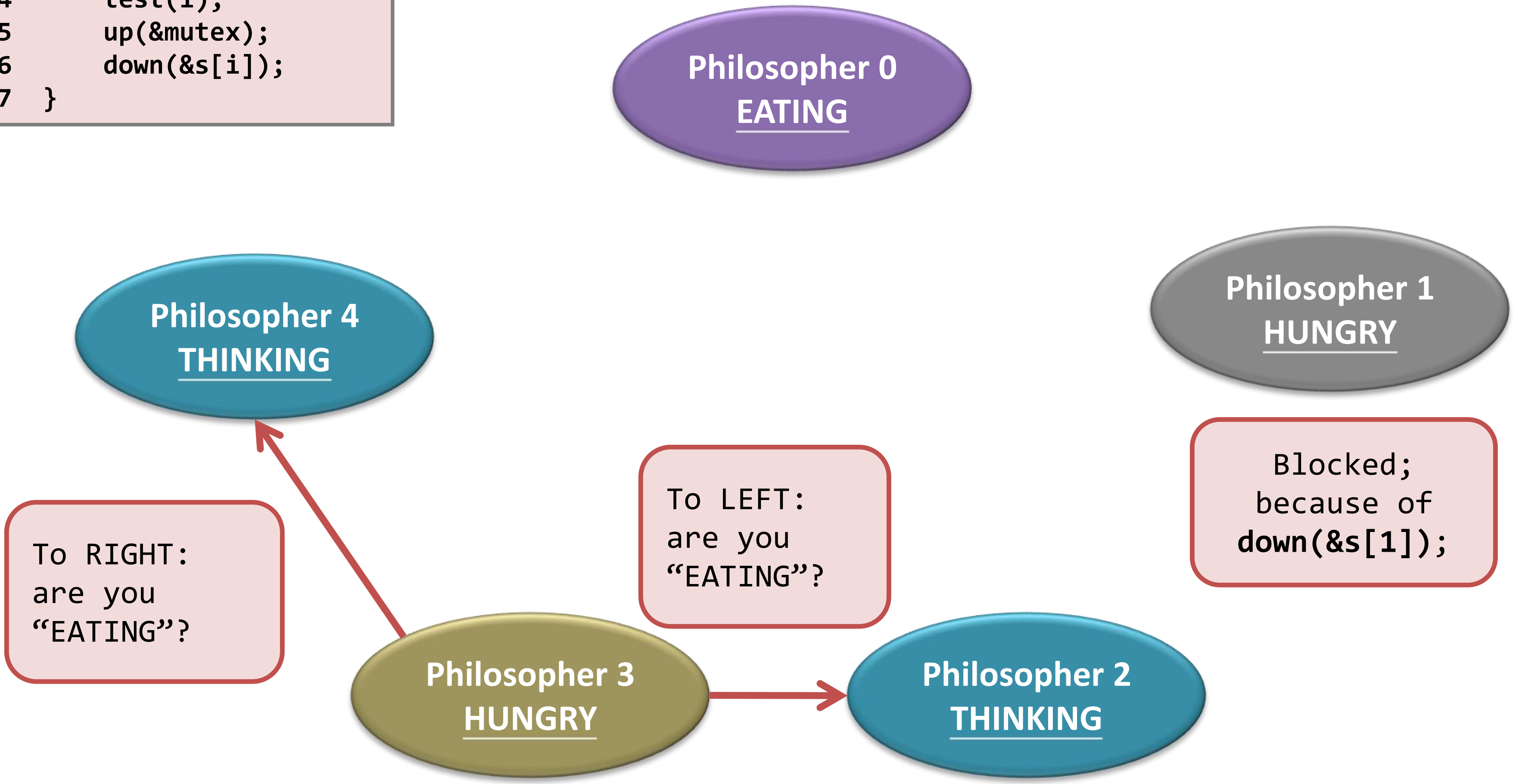
```
void take(int i) {
      down(&mutex);
      state[i] = HUNGRY;
      test(i);
      up(&mutex);
      down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

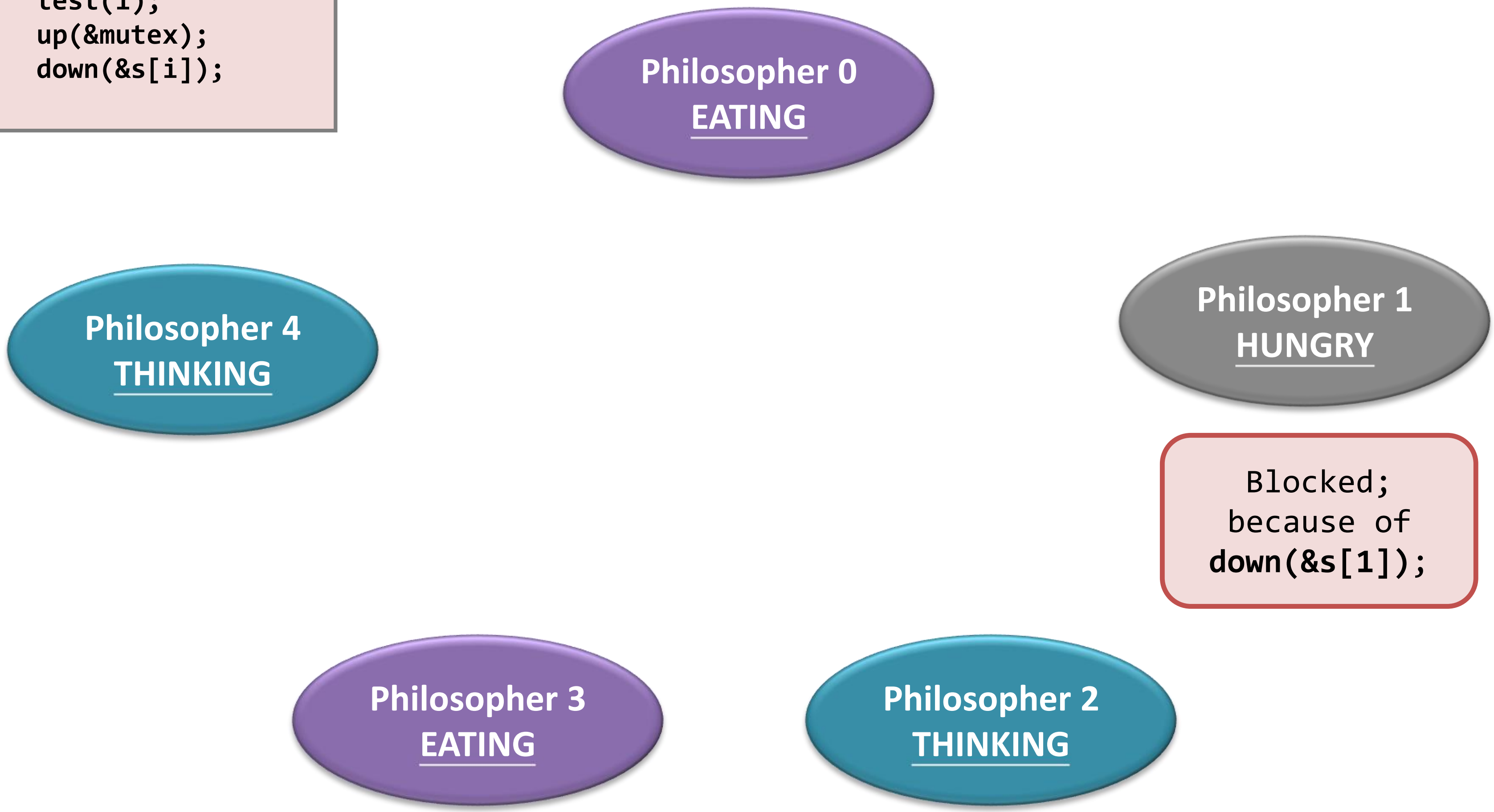
```
void take(int i) {
      down(&mutex);
      state[i] = HUNGRY;
      test(i);
      up(&mutex);
      down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

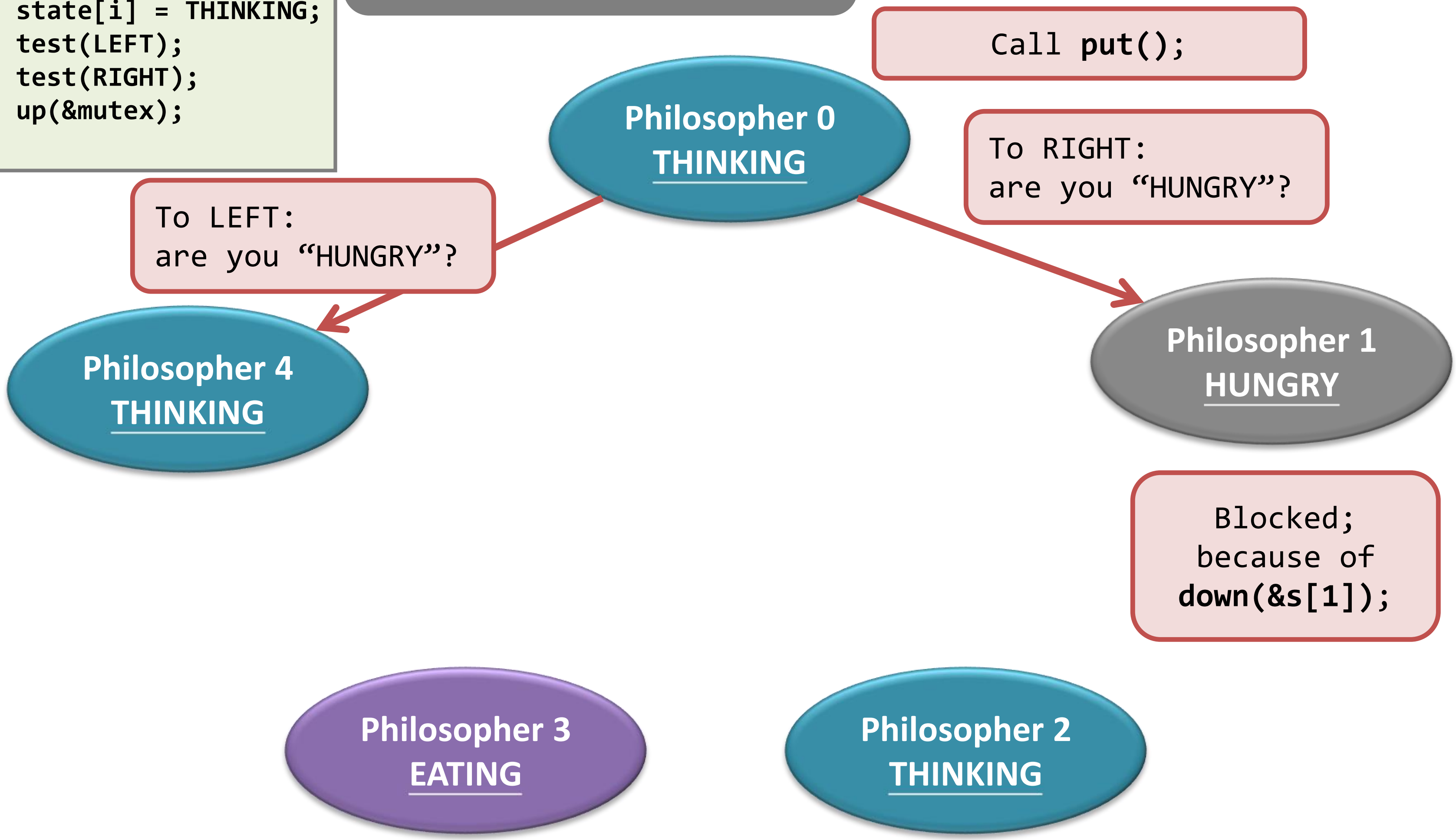


Dining philosopher – the final solution.

Section exit

```
void put(int i) {
     down(&mutex);
     state[i] = THINKING;
     test(LEFT);
     test(RIGHT);
     up(&mutex);
}
```

An illustration: How can Philosopher 1 start eating?

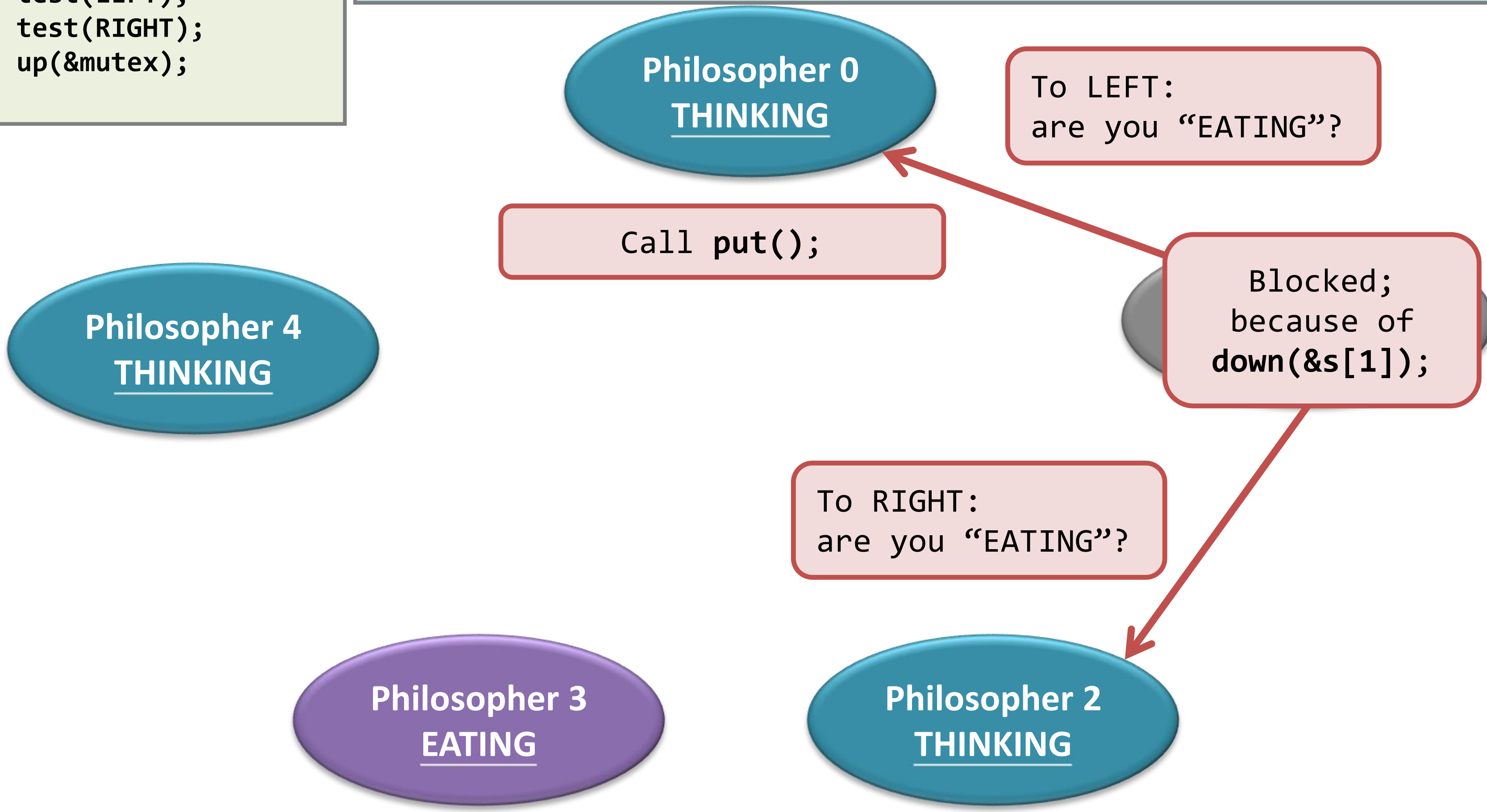


Dining philosopher – the final solution.

Section exit

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

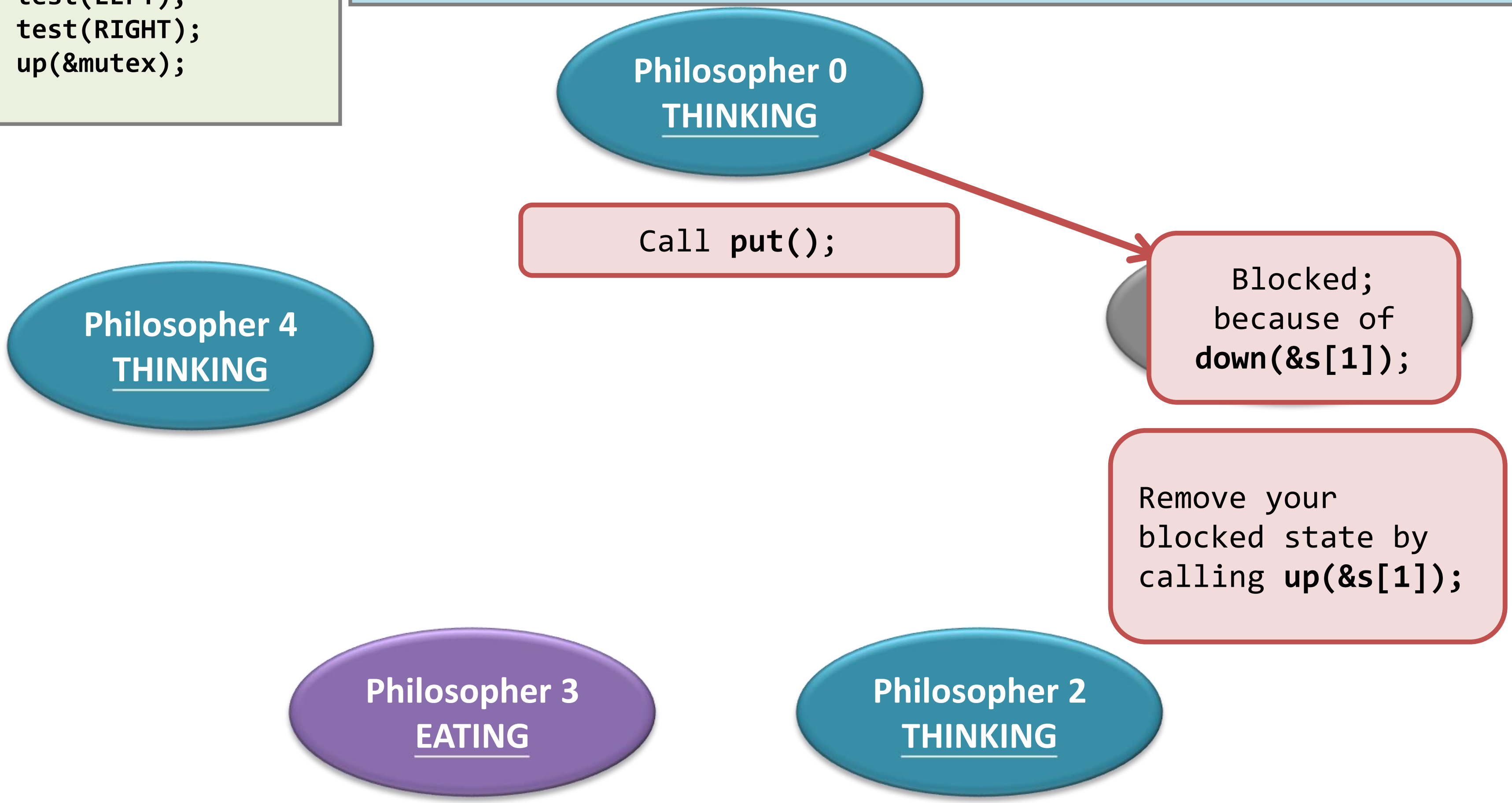


Dining philosopher – the final solution.

Section exit

```
void put(int i) {
      down(&mutex);
      state[i] = THINKING;
      test(LEFT);
      test(RIGHT);
      up(&mutex);
}
```

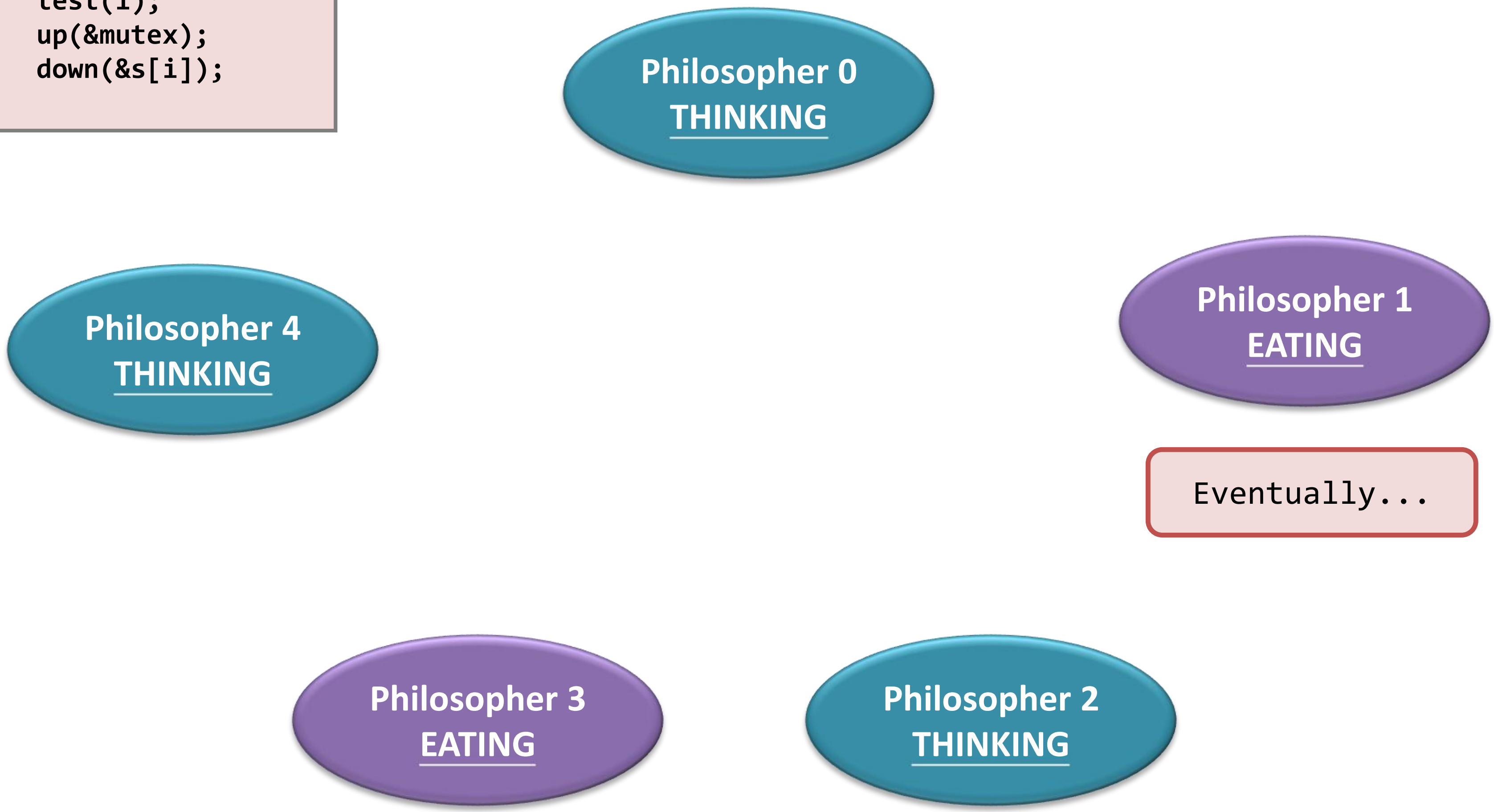
```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```



Dining philosopher – the final solution.

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```



Dining philosopher - summary

- What is the shared object in the final solution?
  - How to guarantee the mutual exclusion

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Section exit

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

Dining philosopher - summary

- Think:
  - Why the semaphore s[N] is needed
  - How to set its initial value

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Dining philosopher - summary

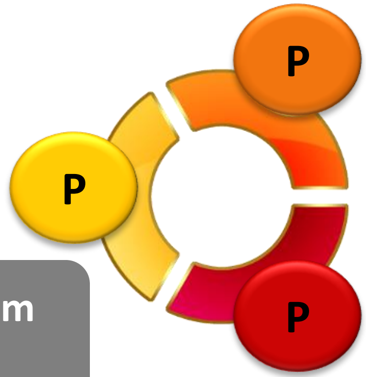
- Solution to IPC problem can be difficult to comprehend.
  - Usually, intuitive methods failed.
  - Depending on time, e.g., sleep(1), does not guarantee a useful solution.

- As a matter of fact, dining philosopher **is not restricted to 5 philosophers**.

The Deadlock Problem

Classic IPC problems

- Dining philosopher problem
- **Producer-consumer problem**
- Reader-writer problem

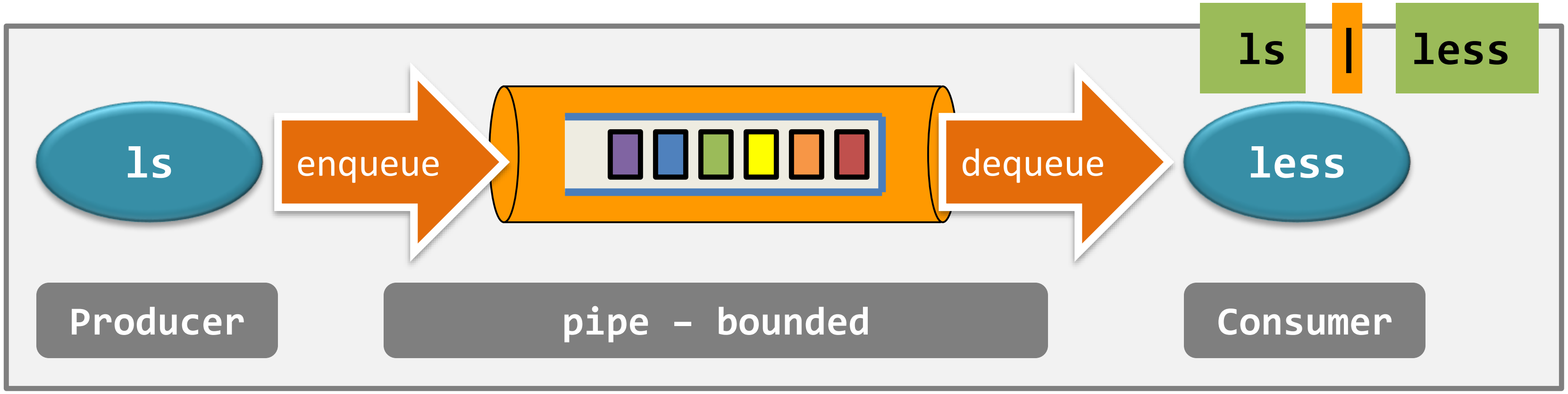


Let’s teach them not to fight.

Producer-consumer problem – recall

- Also known as the **bounded-buffer problem**.

A bounded buffer	-It is a shared object; -Its size is bounded, say N slots. -It is a queue (imagine that it is an array implementation of queue).
A producer process	-It produces a unit of data, and -writes that a piece of data to the tail of the buffer at one time.
A consumer process	-It removes a unit of data from the head of the bounded buffer at one time.



Producer-consumer problem – recall

Producer-consumer requirement #1	When the **producer** wants to (a) put a new item in the buffer, but (b) **the buffer is already full**… Then, (1) **The producer should be suspended**, and (2) **The consumer should wake the producer up** after she has dequeued an item.
Producer-consumer requirement #2	When the **consumer** wants to (a) consumes an item from the buffer, but (b) **the buffer is empty**… Then, (1) **The consumer should be suspended**, and (2) **The producer should wake the consumer up** after she has enqueued an item.

Producer-consumer problem

- **Pipe is working fine. Is it enough?**
  - What if we cannot use pipes?
    - Say, there are 2 producers and 2 consumers without any parent-child relationships?
  - Then, **the kernel can't protect you with a pipe.**

- In the following, we revisit the producer-consumer problem with **the use of shared objects and semaphores**, instead of pipe.

Design – Semaphores

- **ISSUE #1: Mutual Exclusion**.

> **Solution:** one binary semaphore (mutex)

- **ISSUE #2: Synchronization (coordination)**.
  - Remember the two requirements:
    - Insert an item when it is not FULL
    - Consume an item when it is not EMPTY
  - Can we use a binary semaphore?

> **Solution:** two counting semaphores (full & empty)

Producer-consumer problem – solution

<u>Note</u>

The functions “**insert_item()**” and “**remove_item()**” are accessing the bounded buffer (codes in critical section).

The size of the bounded buffer is “**N**”.

Producer function

```
void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        insert_item(item);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;
    while(TRUE) {
        item = remove_item();
        consume_item(item);
    }
}
```

Producer-consumer problem – solution

Note

Mutual exclusion requirement

Synchronization requirement

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        insert_item(item);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        item = remove_item();
        consume_item(item);
    }
}
```

Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();


        insert_item(item);


    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {


        item = remove_item();


        consume_item(item);
    }
}
```

Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

mutex:
What is its purpose?
Why is the initial value of mutex 1?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();

        down(&mutex);
        insert_item(item);
        up(&mutex);

    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {

        down(&mutex);
        item = remove_item();
        up(&mutex);

        consume_item(item);
    }
}
```

Producer-consumer problem – Understanding

Why we need three semaphores, "empty", "full", "mutex"?

mutex:
what is its purpose?
Why is the initial value of mutex 1?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();

        down(&mutex);
        insert_item(item);
        up(&mutex);

    }
}
```

The "**mutex**" stands for mutual exclusion.

- **down()** and **up()** statements are the entry and the exit of the critical section, respectively.

What is the meaning of the initial value 1?

Producer-consumer problem – Understanding

Why we need three semaphores, “<u>empty</u>”, “<u>full</u>”, “<u>mutex</u>”?

How about “full” and “empty”?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer-consumer problem – Understanding

- The two variables are not for mutual exclusion, but for **process synchronization**.
  - "*Process synchronization*" means **to coordinate** the set of processes so as to produce meaningful output.

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer-consumer problem – Understanding

For "**empty**",
- Its initial value is N;
- It decrements by 1 in each iteration.
- When it reaches 0, the producers sleeps.

So, does it sound like one of the requirements?

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

The consumer wakes the producer up when it finds "**empty**" is 0.

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer-consumer problem – Understanding

- Semaphore can be more than mutual exclusion!

`empty`	It represents the number of empty slots.
`full`	It represents the number of occupied slots.

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer-consumer problem – question

Question.
Can we swap Lines 6 & 7 of the producer?

Let us simulate what will happen with the modified code!

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

Producer function

```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          down(&mutex);
7*          down(&empty);
8           insert_item(item);
9           up(&mutex);
10          up(&full);
11      }
12  }
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer-consumer problem – question

We are showing the value of the semaphores before the producer is suspended.

mutex = 1 | empty = 0 | full = N

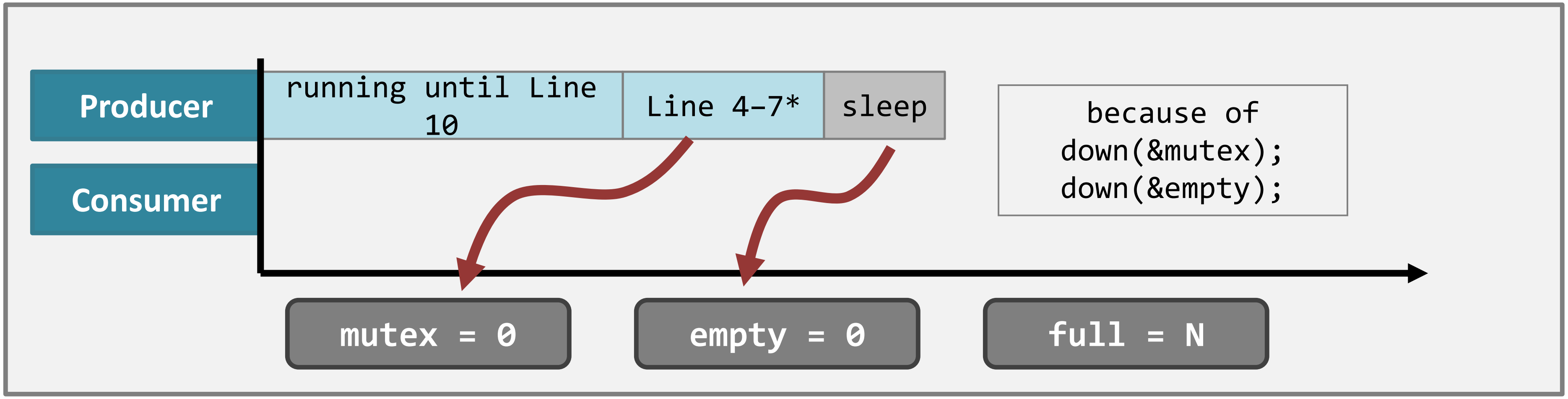
Producer function

```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          down(&mutex);
7*          down(&empty);
8           insert_item(item);
9           up(&mutex);
10          up(&full);
11      }
12  }
```

Consumer Function

```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           down(&full);
6           down(&mutex);
7           item = remove_item();
8           up(&mutex);
9           up(&empty);
10          consume_item(item);
11      }
12  }
```

Producer-consumer problem – question



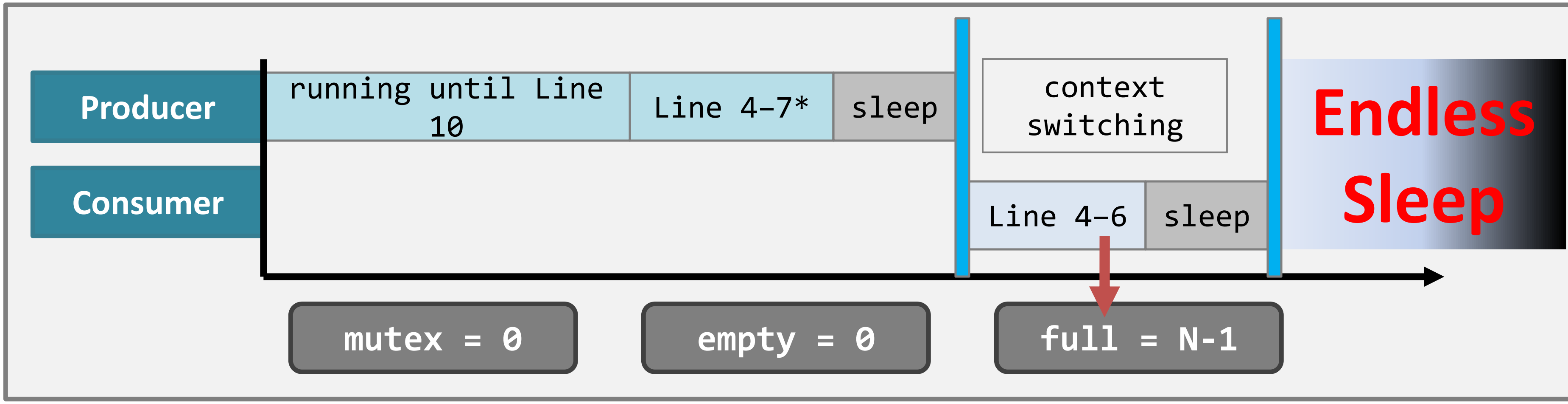
Producer function

```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          down(&mutex);
7*          down(&empty);
8           insert_item(item);
9           up(&mutex);
10          up(&full);
11      }
12  }
```

Consumer Function

```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           down(&full);
6           down(&mutex);
7           item = remove_item();
8           up(&mutex);
9           up(&empty);
10          consume_item(item);
11      }
12  }
```

Producer-consumer problem – question



Producer function

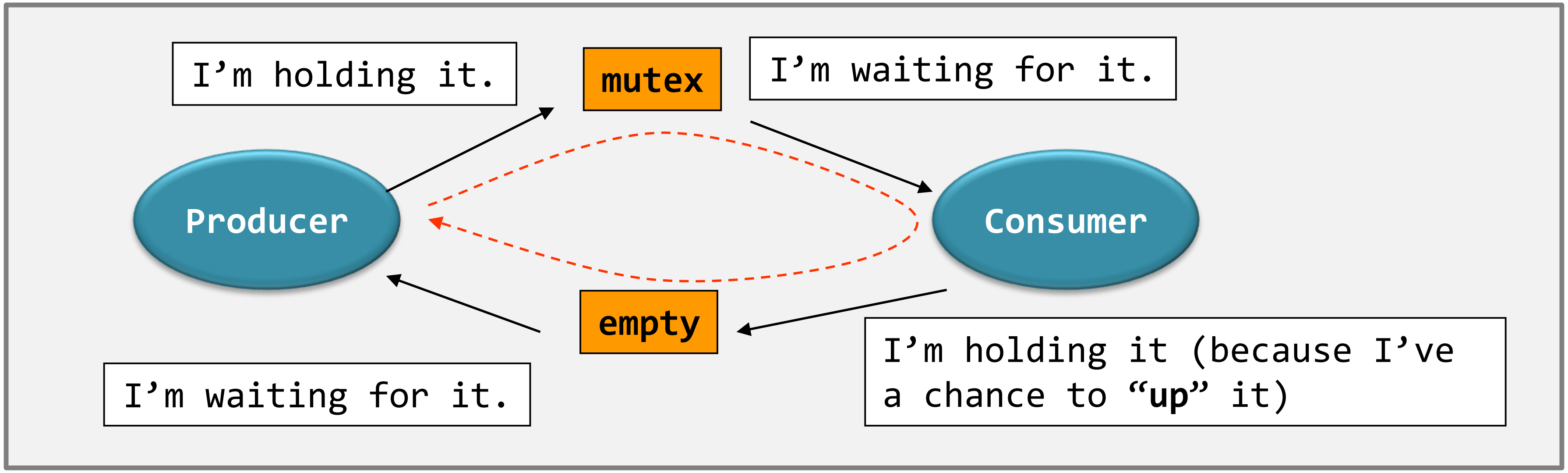
```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          down(&mutex);
7*          down(&empty);
8           insert_item(item);
9           up(&mutex);
10          up(&full);
11      }
12  }
```

Consumer Function

```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           down(&full);
6           down(&mutex);
7           item = remove_item();
8           up(&mutex);
9           up(&empty);
10          consume_item(item);
11      }
12  }
```

Producer-consumer problem

- **Deadlock** happens when a **circular wait** appears
  - The producer is waiting for the consumer to "**`up()`**" the "**`empty`**" semaphore, and
  - the consumer is waiting for the producer to "**`up()`**" the "**`mutex`**" semaphore.



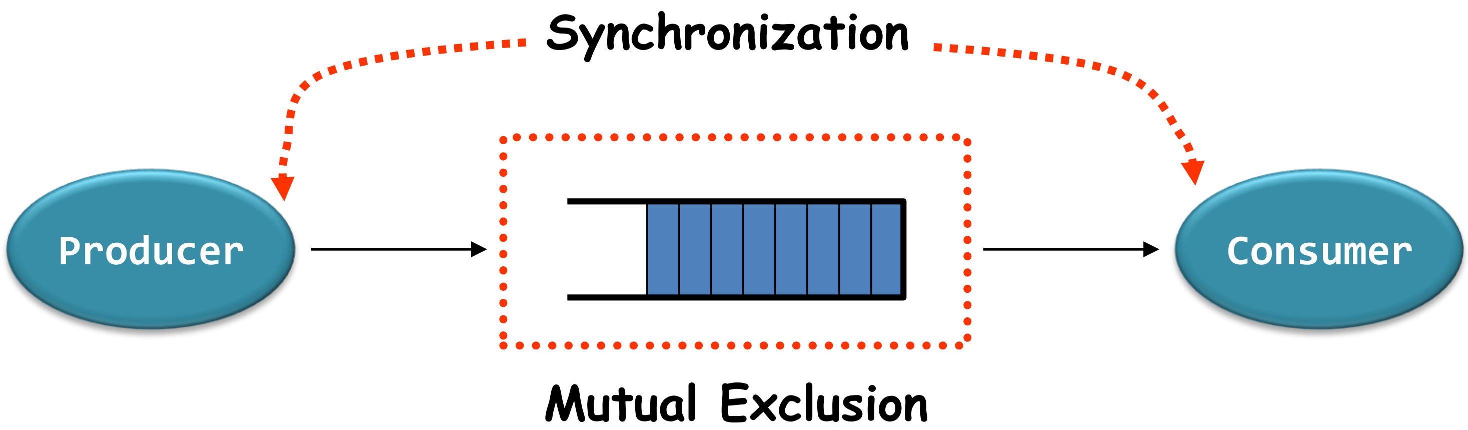
Producer-consumer problem

- **Deadlock** happens when a **circular wait** appears
  - The producer is waiting for the consumer to "`up()`" the "`empty`" semaphore, and
  - the consumer is waiting for the producer to "`up()`" the "`mutex`" semaphore.

- **No progress could be made by all processes + All processes are blocked.**
  - **Implication**: careless implementation of the producer-consumer solution can be disastrous.

Summary on producer-consumer problem

- The problem can be divided into two sub-problems.
  - Mutual exclusion.
    - The buffer is a shared object. Mutual exclusion is needed.
  - Synchronization.
    - Because the buffer's size is bounded, coordination is needed.



Summary on producer-consumer problem

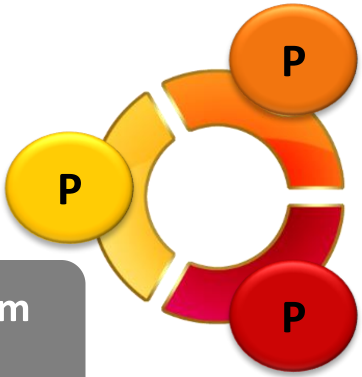
- How to guarantee mutual exclusion?
  - A **binary semaphore** is used as the entry and the exit of the critical sections.

- How to achieve synchronization?
  - Two semaphores are used as **counters** to monitor the status of the buffer.
  - Two semaphores are needed because the two suspension conditions are different.

The Deadlock Problem

Classic IPC problems

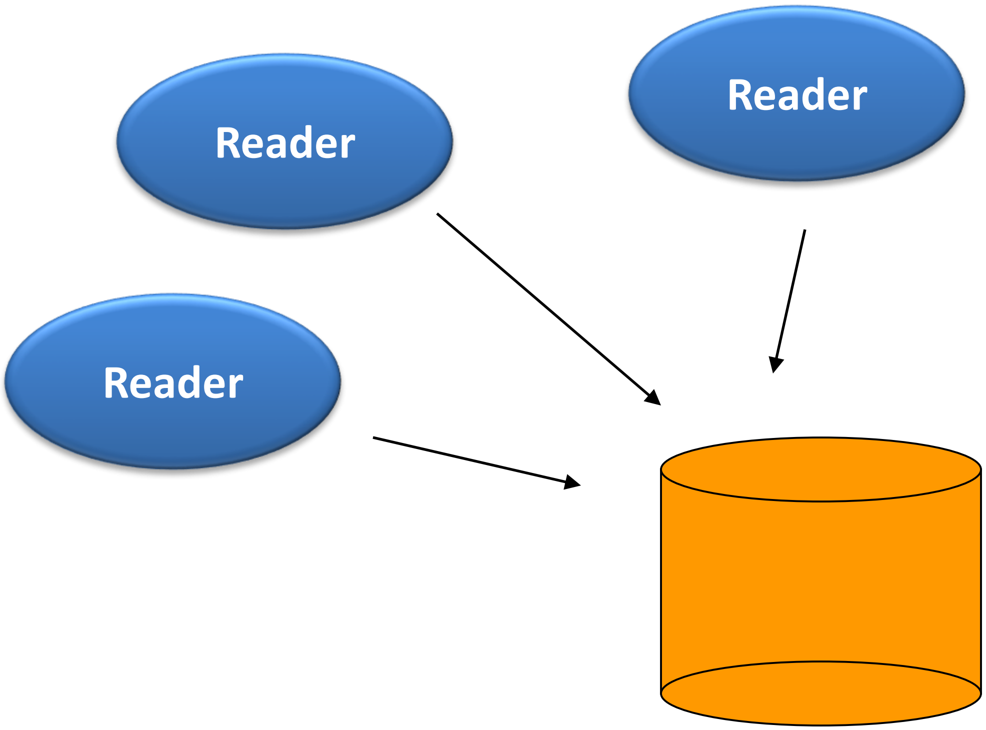
- Dining philosopher problem
- Producer-consumer problem
- **Reader-writer problem**



Let’s teach them not to fight.

Reader-writer problem – introduction

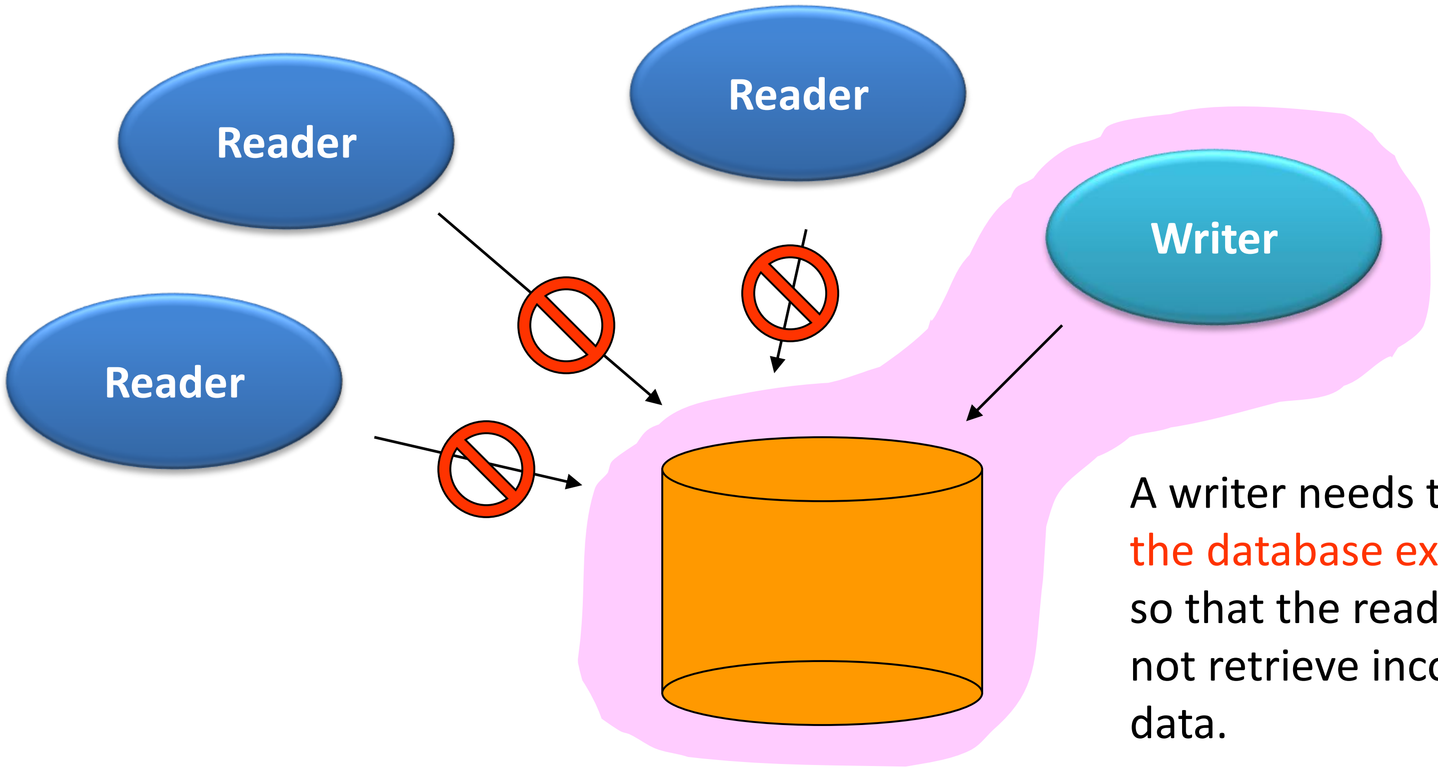
- It is a concurrent database problem.



Readers are
allowed to read
the content of the
database concurrently.

Reader-writer problem – introduction

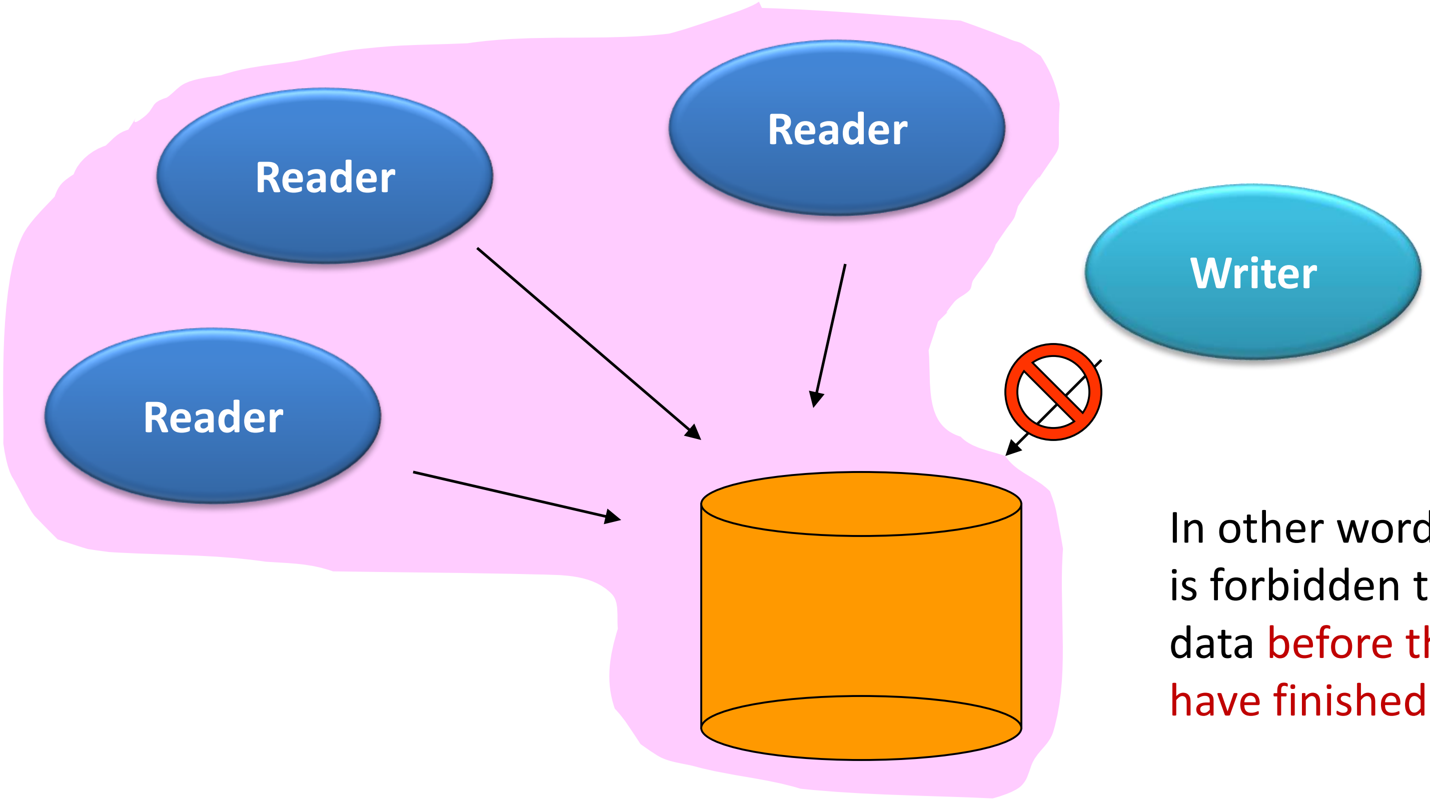
- It is a concurrent database problem.



A writer needs to lock the database exclusively so that the readers would not retrieve inconsistent data.

Reader-writer problem – introduction

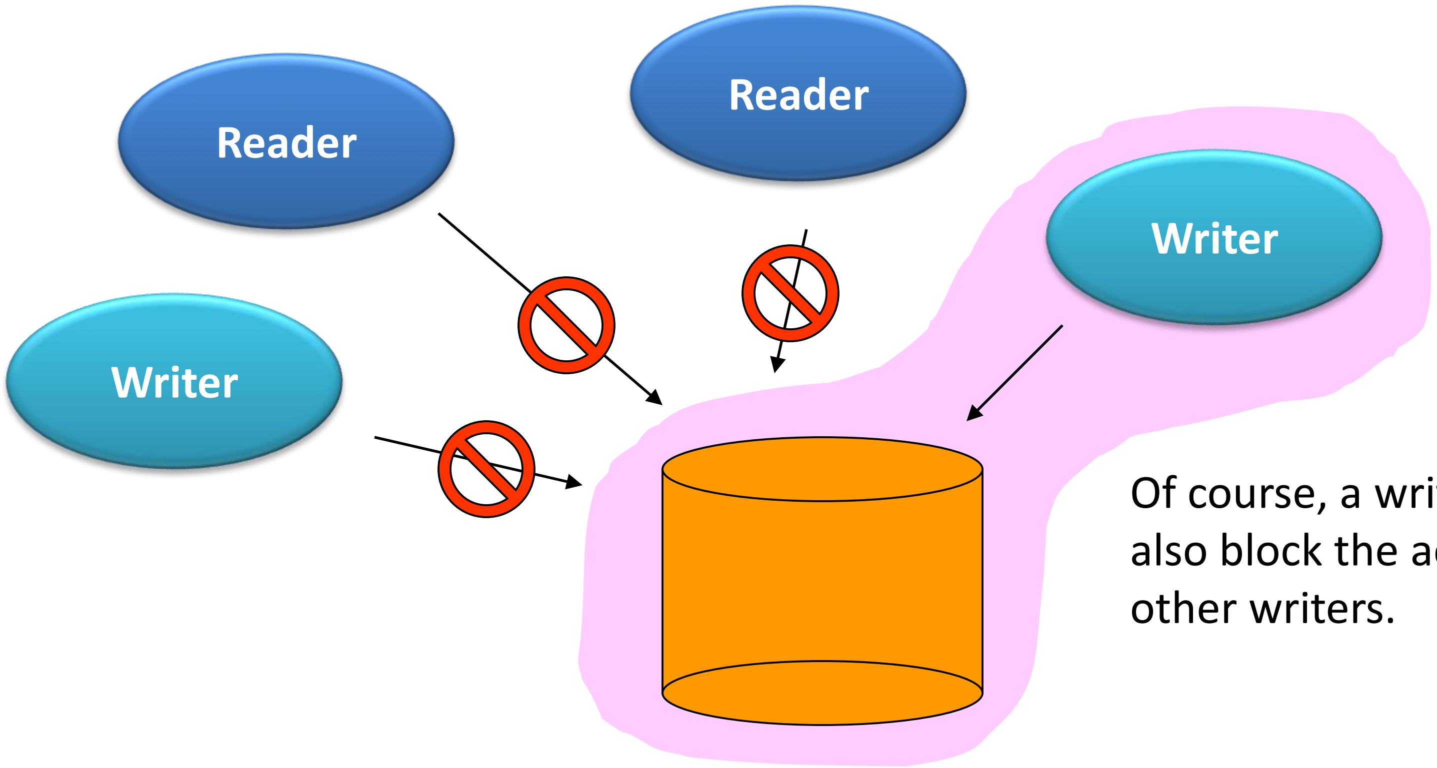
- It is a concurrent database problem.



In other words, a writer is forbidden to write any data before the readers have finished reading.

Reader-writer problem – introduction

- It is a concurrent database problem.



Of course, a writer will also block the access from other writers.

Reader-writer problem – subproblems

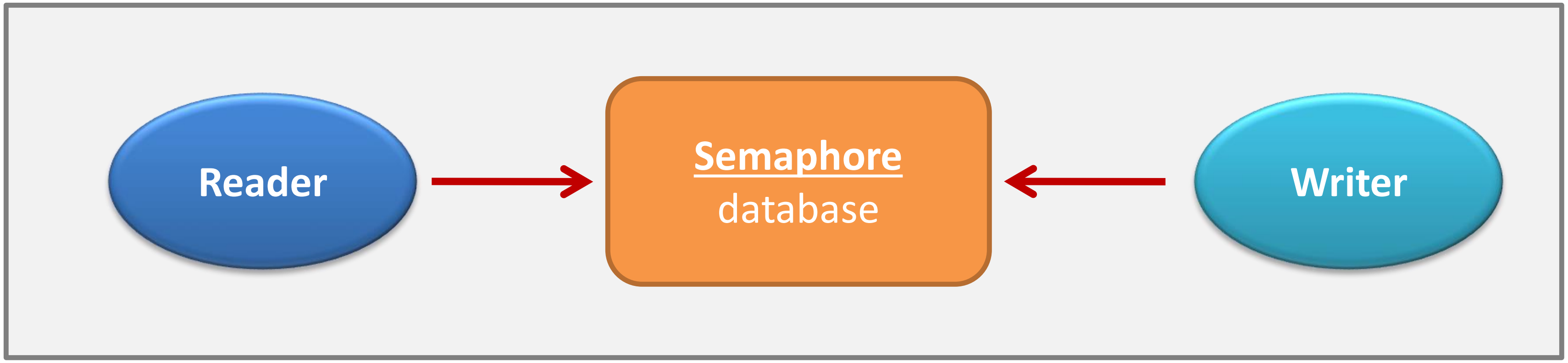
- A mutual exclusion problem.
  - The database is a shared object.

- A synchronization problem.
  - **Rule 1.** While a reader is reading, other readers is allowed to read the database.
  - **Rule 2.** While a reader is reading, no writers is allowed to write to the database.
  - **Rule 3.** While a writer is writing, no writers and readers are allowed to access the database.

- A concurrency problem.
  - **Simultaneous access for multiple readers** is allowed and must be guaranteed.

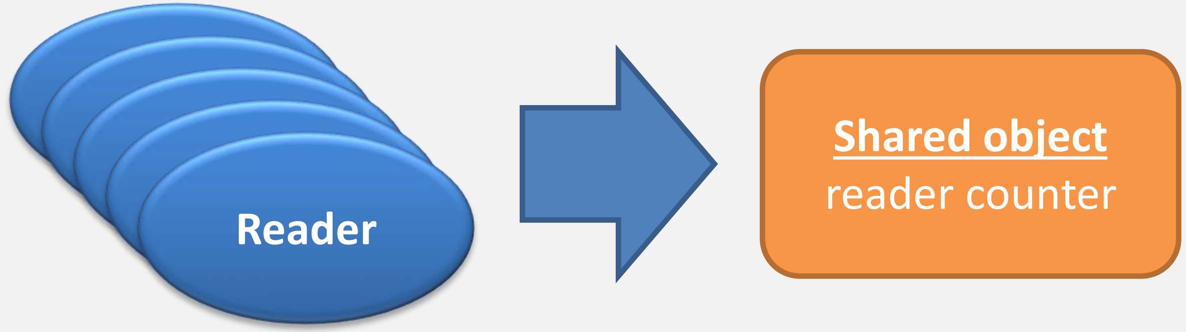
Reader-writer problem – solution outline

- **Mutual exclusion**: relate the readers and the writers to one semaphore.
  - This guarantees **no readers and writers** could proceed to their critical sections at the same time.
  - This also guarantees **no two writers** could proceed to their critical sections at the same time.



Reader-writer problem – solution outline

- **Readers' concurrency**
  - The **first reader coming** to the system "**`down()`**" the "**`database`**" semaphore.
  - The **last reader leaving** the system "**`up()`**" the "**`database`**" semaphore.



Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;
semaphore mutex = 1;
int read_count  = 0;
```

Writer function

```
void writer(void) {
    while(TRUE) {
        prepare_write();
        down(&db);
        write_database();
        up(&db);
    }
}
```

Section Entry: `prepare_write(); down(&db);`

Critical Section: `write_database();`

Section Exit: `up(&db);`

Reader Function

```
void reader(void) {
    while(TRUE) {
        down(&mutex);
        read_count++;
        if(read_count == 1)
            down(&db);
        up(&mutex);
        read_database();
        down(&mutex);
        read_count--;
        if(read_count == 0)
            up(&db);
        up(&mutex);
        process_data();
    }
}
```

Section Entry: `down(&mutex); read_count++; if(read_count == 1) down(&db); up(&mutex);`

Critical Section: `read_database();`

Section Exit: `down(&mutex); read_count--; if(read_count == 0) up(&db); up(&mutex);`

Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;
semaphore mutex = 1;
int read_count  = 0;
```

Guarantee the mutual exclusion between the readers and the writers.

Protect the “**read_count**” variable.

Keep track of the number of readers in the system.

Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;
semaphore mutex = 1;
int read_count  = 0;
```

Writer function

```
void writer(void) {
    while(TRUE) {
        prepare_write();
        down(&db);

        write_database();

        up(&db);
    }
}
```

Section Entry: `prepare_write(); down(&db);`

Critical Section: `write_database();`

Section Exit: `up(&db);`

The writer is allowed to enter its critical section when no other process is in its critical section (protected by the “**db**” semaphore)

Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;
semaphore mutex = 1;
int read_count  = 0;
```

The first reader "**down()**" the "**db**" semaphore so that no writers would be allowed to enter their critical sections.

The last reader "**up()**" the "**db**" semaphore so as to let the writers to enter their critical section.

Reader Function

```
void reader(void) {
    while(TRUE) {
        down(&mutex);
        read_count++;
        if(read_count == 1)
            down(&db);
        up(&mutex);

        read_database();

        down(&mutex);
        read_count--;
        if(read_count == 0)
            up(&db);
        up(&mutex);
        process_data();
    }
}
```

Reader-writer problem – summary

- This solution does not limit the number of readers and the writers admitted to the system.
  - A realistic database needs this property.

- This solution gives readers a higher priority over the writers.
  - Whenever there are readers, writers must be blocked, not the other way round.

- **What if a writer should be given a higher priority?**

Summary on IPC problems

- The problems have the following properties in common:
  - Multiple processes;
  - Shared and limited resources;
  - Processes have to be synchronized in order to generate useful output;

- The synchronization algorithms have the following requirements in common:
  - Guarantee mutual exclusion;
  - Uphold the correct synchronization among processes;
  - Deadlock-free.

Summary on Ch5

Cooperating Processes
Processes Communication

→

IPC methods
Shared memory, Pipes, Sockets

Race Condition
Mutual Exclusion

→

How to realize
Define **critical section**

→

How to implement
☐ 4 requirements & 5 schemes
☐ **Semaphore**

Process Synchronization
Deadlock

→

Classic problems
☐ Producer-consumer problem
☐ Dining philosopher problem
☐ Reader-writer problem